Time-Predictable Communication on a Time-Division Multiplexing Network-on-Chip Multicore

Sørensen, Rasmus Bo

Publication date:
2016

Document Version
Publisher's PDF, also known as Version of record

Time-Predictable Communication on a Time-Division Multiplexing Network-on-Chip Multicore

Rasmus Bo Sørensen

Kongens Lyngby 2016
PhD-2016-423

Technical University of Denmark
Department of Applied Mathematics and Computer Science
Richard Petersens Plads, building 324
2800 Kongens Lyngby, Denmark
Phone +45 4525 3031
compute@compute.dtu.dk
www.compute.dtu.dk PhD-2016-423

Abstract

This thesis presents time-predictable inter-core communication on a multicore platform with a time-division multiplexing (TDM) network-on-chip (NoC) for hard real-time systems. The thesis is structured as a collection of papers that contribute within the areas of: reconfigurable TDM NoCs, static TDM scheduling, and time-predictable inter-core communication.

More specifically, the work presented in this thesis investigates the interaction between hardware and software involved in time-predictable inter-core communication on the multicore platform. The thesis presents: a new generation of the Argo NoC network interface (NI) that supports instantaneous reconfiguration, a TDM traffic scheduler that generates virtual circuit (VC) configurations for the Argo NoC, and software functions for two types of inter-core communication.

The new generation of the Argo NoC adds the capability of instantaneously reconfiguring VCs and it addresses the identified shortcomings of the previous generation. The VCs provide the guaranteed bandwidth and latency required to implement time-predictable inter-core communication on top of the Argo NoC. This new Argo generation is, in terms of hardware, less than half the size of NoCs that provide similar functionalities and it offers a higher degree of flexibility to the application programmer.

The developed TDM scheduler supports a generic TDM NoC and custom parameterizable communication patterns. These communication patterns allow the application programmer to generate schedules that provide a set of VCs that efficiently uses the hardware resources. The TDM scheduler also shows better results, in terms of TDM period, compared to previous state-of-the-art TDM schedulers. Furthermore, we provide a description of how a communication pattern can be optimized in terms of shortening the TDM period.

The thesis identifies two types of inter-core communication that are commonly used in real-time systems: message passing and state-based communica-

tion. We implement message passing as a circular buffer with the data transfer through the NoC. The worst-case execution time (WCET) of the send and receive functions of our implementation is not dependent on the message size. We also implement five algorithms for state-based communication and analyze them in terms of the WCET and worst-case communication delay. The five algorithms each have scenarios where they are better than the others.

This thesis shows in detail how time-predictable inter-core communication can be implemented in an efficient way, from the low-level hardware to the high-level software functions.

Resumé

Denne afhandling præsentere tidsforudsigelig kommunikation mellem processor kerner på en multikerne platform med et time-division multiplexing (TDM) network-on-chip (NoC) for hårde tidstro systemer. Afhandlingen er struktureret som en samling af videnskabelige artikler, der bidrager inden for områderne: rekonfigurerbare TDM NoC'er, statiske TDM planlæggere og tidsforudsigelig kommunikation mellem processor kerner.

Mere specifikt udforsker denne afhandling sammenspillet mellem hardware og software der er involveret i tidsforudsigelig kommunikation mellem processor kerner på en multikerne platform. Denne afhandling præsenterer: en ny generation af Argo NoC'ets network interface (NI) der supporterer øjeblikkelig re-konfigurering, en TDM trafik planlægger der generere virtual circuit (VC) konfigurationer for Argo NoC'et og software funktioner for to typer af kommunikation mellem processor kerner.

Den nye generation af Argo NoC'et tilføjer funktionalitet til øjeblikkeligt re-konfigurering af VCs og den adresserer de mangler vi har identificeret ved den tidligere generation. VC'erne sikre de garanteret båndbredde og latenstids krav for at implementere tidsforudsigelig kommunikation mellem kerner på Argo NoC'et. Denne nye generation af Argo er, i forhold til hardware størrelse, halvt så lille i forhold til NoC'er med lignende funktionalitet. Endvidere tilbyder den nye generation et højere niveau af fleksibilitet til applikations programmøren.

Den udviklede TDM planlægger supporterer en generisk TDM NoC og brugertilpasset kommunikationsmønstre. Disse kommunikationsmønstre tillader applikations programmøren at generere planer hvor VC'er gør effektivt brug af hardware ressourcerne. TDM planlæggeren kan generere planer med kortere perioder sammenlignet med tidligere publicerede TDM planlæggere. Ydermere giver vi en beskrivelse af hvordan et kommunikations mønster kan optimeres med hensyn til forkortelse af TDM perioden.

Denne afhandling identificere to typer af kommunikation mellem processor kerner der ofte bruges i tidstro systemer: message passing og tilstandsbaseret kommunikation. Vi implementerer message passing som en cirkulær buffer med data overførslen gennem NoC'en. For vores implementering er worst-case execution time (WCET) for en send og modtag funktionerne ikke afhængige af meddelelsens størrelse. Vi implementere også fem algoritmer for tilstandsbaseret kommunikation og analysere dem med hensyn til WCET og worst-case communication delay. De fem algoritmer har hver deres fordele i forskellige scenarier.

Denne afhandling viser i detaljen hvordan tidsforudsigelig kommunikation mellem processor kerner kan implementeres på en effektiv måde, fra den lavniveau hardware til de højniveau software funktioner.

Preface

This thesis was prepared at DTU Compute in fulfillment of the requirements for acquiring a Ph.D. degree. This work was funded by the Danish Council for Independent Research | Technology and Production Sciences, Project no. 12-127600: Hard Real-Time Embedded Multiprocessor Platform (RTEMP) project.

The work was supervised by Professor Jens Sparsø and Associate Professor Martin Schoeberl and was carried out in the context of the T-CREST platform. As the RTEMP project builds on the results of the T-CREST project, the source code developed during the RTEMP project is publicly available under the BSD open source license at the Github page of the T-CREST project.[1]

During my Ph.D. studies, I stayed one and a half months at the aerospace department of GMV in Lisbon, Portugal, to study how aerospace systems are structures and what they require from a hardware platform.

The thesis is based on a collection of submitted or published papers that are listed in the introduction chapter. The thesis deals with time-predictable inter-core communication on a time-division multiplexing network-on-chip multicore platform for hard real-time systems.

The thesis contributes within three topics that are motivated and introduced in the first chapter. The following 5 chapters are five scientific papers that contribute within the three topics and the last chapter concludes the thesis.

Lyngby, 27-July-2016

Rasmus Bo Sørensen

[1] T-CREST Github project page: `https://github.com/t-crest`

Acknowledgements

This thesis marks the end of my academic career. My time in academia has felt like solving a puzzle without being able to see or feel the pieces. Even though this has been frustrating at times, I have enjoyed solving the encountered problems and the challenge of explaining why my work is worthy of publishing.

First and foremost, I would like to thank my supervisor Professor Jens Sparsø for his encouragement, inspiring discussions, and computer history lessons and also my co-supervisor Associate Professor Martin Schoeberl for continuously challenging my view of the big picture.

Additionally, I would also like to thank my colleagues Luca Pezzarossa, Wolfgang Puffitsch, Christoph Thomas Müller, Torur Biskopstø Strøm, and Evangelia Kasapaki for all the interesting and fruitful discussions we have had, some of which had no direct relation to our work, but nonetheless very interesting and inspiring.

Furthermore, I would like to thank Steen Nielsen and Kurt Müller from Danfoss for their hospitality during my stays in Gråsten and their collaboration during the RTEMP project. I would also like to thank all the people at GMV in Lisboa for their hospitality and for making my stay pleasant and educative. It was very interesting to get a taste of how things work in industry, which in many cases feels like a parallel universe to academia.

Finally, I would like to thank my family for their love and long lasting support. I would not be where I am today without their encouragement. Also a special thanks to Tabita Niemann Kristensen for bearing with me and listing to all my frustrations, I think that these past 3 years was harder on her than it was on me.

Contents

List of Acronyms

ACET	Average-case execution time
ALNS	Adaptive large neighborhood search
API	Application programming interface
BE	Best effort
CISC	Complex instruction set computer
DMA	Direct memory access
FFT	Fast Fourier transform
FIFO	First in first out
FPGA	Field programmable gate-array
GRASP	Greedy randomized adaptive search procedure
GS	Guaranteed service
MPI	Message passing interface
NI	Network interface
NoC	Network-on-Chip
RAM	Random access memory
RISC	Reduced instruction set computer
ROM	Read only memory

RTL	Register transfer level
SPM	Scratch pad memory
SRAM	Static random access memory
TDM	Time-division multiplexing
VC	Virtual circuit
WCCD	Worst-case communication delay
WCET	Worst-case execution time

List of Publications

Papers included in this thesis:

1. Rasmus Bo Sørensen, Luca Pezzarossa, and Jens Sparsø,
"An Area-Efficient TDM NOC Supporting Reconfiguration for Mode Changes",
10th IEEE/ACM International Symposium on Networks-on-Chip (NOCS),
2016.

2. Rasmus Bo Sørensen, Luca Pezzarossa, Martin Schoeberl and Jens Sparsø,
"A Resource-Efficient Network Interface Supporting Low Latency Reconfiguration of Virtual Circuits in Time-Division Multiplexing Networks-on-Chip", *Submitted to Journal of Systems Architecture*, 2016.

3. Rasmus Bo Sørensen, Jens Sparsø, Mark Ruvald Pedersen, and Jaspur Højgaard,
"A Metaheuristic Scheduler For Time Division Multiplexed Networks-on-Chip", *10th Workshop on Software Technologies for Future Embedded and Ubiquitous Systems (SEUS)*, pages 309–316, 2014.

4. Rasmus Bo Sørensen, Wolfgang Puffitsch, Martin Schoeberl, and Jens Sparsø,
"Message Passing on a Time-predictable Multicore Processor", *IEEE 18th Symposium on Real-time Computing (ISORC)*, pages 51-59, 2015.

5. Rasmus Bo Sørensen, Martin Schoeberl, and Jens Sparsø,
"State-based Communication on Time-predictable Multicore Processors",
Submitted to International Conference on Real-Time Networks and Systems (RTNS), 2016.

Additional publications:

6. Rasmus Bo Sørensen, Martin Schoeberl, and Jens Sparsø, "A light-weight statically scheduled network-on-chip", *NORCHIP*, pp. 1–6, 2012.

7. Evangelia Kasapaki, Jens Sparsø, Rasmus Bo Sørensen, and Kees Goossens, "Router designs for an asynchronous time-division-multiplexed network-on-chip", *Euromicro Conference on Digital System Design (DSD)*, pp. 319–326, 2013.

8. Christoph Thomas Müller, Evangelia Kasapaki, Rasmus Bo Sørensen, and Jens Sparsø, "Synthesis and layout of an asynchronous network-on-chip using standard EDA tools", *NORCHIP*, pp. 1–6, 2014.

9. Ioannis Kotleas, Dean Humphreys, Rasmus Bo Sørensen, Evangelia Kasapaki, Florian Brandner, and Jens Sparsø, "A loosely synchronizing asynchronous router for TDM-scheduled NOCs", *IEEE/ACM International Symposium on Networks-on-Chip (NOCS)*, pp. 151–158, 2014.

10. Martin Schoeberl, Rasmus Bo Sørensen, and Jens Sparsø, "Models of Communication for Multicore Processors", *IEEE International Symposium on Object/Component/Service-Oriented Real-Time Distributed Computing Workshops (ISORCW)*, pp. 9–16, 2015.

11. Wolfgang Puffitsch, Rasmus Bo Sørensen, and Martin Schoeberl, "Time-division multiplexing vs network calculus: a comparison", *Proceedings of the 23rd International Conference on Real Time and Networks Systems (RTNS)*, pp. 289–296, 2015.

12. Luca Pezzarossa, Rasmus Bo Sørensen, Martin Schoeberl, and Jens Sparsø, "Interfacing hardware accelerators to a time-division multiplexing network-on-chip", *Nordic Circuits and Systems Conference (NORCAS): NORCHIP & International Symposium on System-on-Chip (SoC)*, pp. 1–4, 2015.

13. Evangelia Kasapaki, Martin Schoeberl, Rasmus Bo Sørensen, Christoph Müller, Kees Goossens, and Jens Sparsø, "Argo: A Real-Time Network-on-Chip Architecture With an Efficient GALS Implementation", *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, vol. 24, no. 2, pp. 479-492, Feb. 2016.

14. André Rocha, Cláudio Silva, Rasmus Bo Sørensen, Jens Sparsø, and Martin Schoeberl, "Avionics Applications on a Time-predictable Chip-Multiprocessor", *Euromicro International Conference on Parallel, Distributed, and Network-Based Processing (PDP)*, pp. 777–785, 2016.

15. Heiko Falk, Sebastian Altmeyer, Peter Hellinckx, Björn Lisper, Wolfgang Puffitsch, Christine Rochange, Martin Schoeberl, Rasmus Bo Sørensen, Peter Wägemann and Simon Wegener, "TACLeBench: A Benchmark Collection to Support Worst-Case Execution Time Research", *International Workshop on Worst-Case Execution Time Analysis (WCET)*, 2016.

Chapter 1

Introduction

The topic of this thesis is real-time core-to-core communication on a multicore platform using a resource-efficient time-division multiplexing (TDM) network-on-chip (NoC). The thesis contributes in three areas: reconfigurable time-division multiplexing networks-on-chip, static time-division multiplexing scheduling, and time-predictable inter-core communication. The contributions of the three research areas are closely related, because they build on top of one another. The TDM NoC uses the TDM schedules generated by the TDM scheduler to provide guaranteed service (GS) of the NoC data transfers. The GS of the NoC data transfers enable the time-predictable inter-core communication. This chapter presents a motivation of this thesis, the contributions of the author in more detail, and an introduction of the research topics of the remaining chapters of the thesis.

1.1 Motivation

A real-time system is a computing system that is required to deliver a correct response within a predefined time window, often referred to as the deadline. A response that arrives after the deadline, even if it is functionally correct, can lead to failure in the system. If a failure in the systems can have catastrophic consequences, such as loss of human life or damage to the environment, then the real-time system is also a safety-critical system. Therefore, the time-predictability of a real-time system is of paramount importance to ensure correct

behavior of the system. To a large extend, the time-predictability of a real-time application depends on the hardware platform that it is executes on.

After the processor clock frequency stopped increasing in the mid 2000's due to power density limitations, general-purpose processors started moving into the multicore era [11], to continue the increase in overall computational power of a single chip. The adoption of multi/many-core processors in the real-time systems industry is more conservative, but the embedded processors for such real-time systems are also moving into the multi/many-core era [24]. As the development and fabrication costs of application-specific integrated circuits continue to increase, application-specific platforms are only feasible for a few ultra high volume applications. Therefore, we are interested in domain-specific platforms that can support a range of real-time applications.

In recent years, we have seen new commercial domain-specific multi/many-core platforms that also target the real-time domain, such as LEON4 (Gaisler-flex/ESA) [1], P4080 (Freescale) [3], and MPPA-256 (Kalray) [6]. The worst-case off-chip memory access timings of the LEON4 is difficult to calculate[10], because they depend on the behavior of other cores in the system. This makes the LEON4 challenging to use in a hard real-time system. The P4080 is an 8 core platform with the CoreNet fabric interconnecting the 8 cores and the MPPA-256 has 16 clusters of 16 processing cores. Only few hardware details on the P4080 CoreNet fabric are published and for many-core platforms real-time applications need a time-predictable NoC. The NoC of the MPPA-256 has a large hardware size, due to the extensive use of buffers in the routers. We are therefore interested in investigating the resource-efficiency of time-predictable inter-core communication.

The time-predictable NoC that interconnects the cores is one of the challenges of designing a domain-specific many-core platform for real-time systems. The ability to set up GS communication channels is what distinguishes time-predictable NoCs from general-purpose NoCs. A GS communication channel has a guaranteed minimum bandwidth and maximum latency. Implementing GS in NoCs using TDM results in a simple and resource-efficient architecture [12]. TDM avoids buffering in the routers, which is for most NoC architectures the largest contributer towards the area. Alternatives of implementing GS in NoCs are virtual-channels, rate control and link priorities as used by Mango [5] or rate control, buffers and network calculus as used by Kalray MPPA-256 [7]. The Mango NoC and the MPPA-256 NoC have large hardware size and the Æthereal NoC is designed to be application specific [13]. Therefore, we focus on TDM NoCs that provide a high level of flexibility to support an application domain in a resource-efficient way.

The research field of time-predictable NoC architectures, for which we can calculate hard guarantees on latency and bandwidth of inter-core communication at the application level, is still in its infancy and the design space needs to be explored further. Bjerregaard and Mahadevan [4] divides the research on NoCs

into four layers: link layer, network layer, network adapter layer, and system layer. Previous research has targeted NoCs at the network interface layer for real-time systems [33, 13, 23, 29, 32]. To a large extent this research has mainly focused on providing complete solutions, for end-to-end GS communication, directly in the NoC hardware. We want to show that, through a holistic view, a NoC that provides primitive functionality to the software application programming interface (API), results in high flexibility and high resource-efficiency.

The discussion of providing primitives and not solutions in NoCs, can be seen as an analogy to the discussion from the 1980's, where researchers were discussing the instruction set architecture (ISA) of the processor paradigms: complex instruction set computer (CISC) and reduced instruction set computer (RISC). The discussion focused on which processor instructions that a processor should provide to the programmer/compiler. An architecture following the CISC paradigm had a large number of instructions that could execute complex operations in a single instruction. This type of architecture results in dense code, but also, as the name hints, a complex hardware. An architecture following the RISC paradigm only implements the instructions that make up the majority of the execution time of most programs [26] and the complex operations are then implemented using multiple primitive instructions. In the 1980's, a major advantage of the RISC computer was that the reduction in supported instructions reduced the size of the processor enough to make it fit on a single die. The increased level of integration was the main contributing factor to the increased performance of the RISC processors.

As opposed to most of the ISAs of the 1980's that could be programmed in C, the communication interfaces offered to applications by current state-of-the-art NoC architectures have not yet converged into a common set of operations. Therefore, it is difficult to make parallel application that make explicit use of the NoC and can run on a wide range of platforms without modifications. For this reason, it is not straight forward to make a quantitative analysis to determine how to split functionality between hardware and software. To increase the level of integration, we follow the mindset of providing primitives and not solutions, which reduces the area of the NoC hardware and makes room for larger on-chip memories or a larger number of processing cores.

With a focus on flexibility and resource-efficiency this thesis addresses the time-predictability of core-to-core communication crossing the hardware/software interface in each core. In the terminology of Bjerregaard and Mahadevan [4], we investigate the communication stack from the system layer functions in the API down to the hardware components of the network interface layer that sends data packets through the network of routers. We base our work on the T-CREST platform and on the work of Evangelia Kasapaki [17], who focused on the network mechanics and asynchronous routers of the T-CREST platform. Through the gained insight we develop a second generation of the NoC hardware and

investigate how software can implement a time-predictable API on-top of the hardware.

Our goal is to provide cheap and flexible solutions to the application programmer using hardware that offers primitive functions. We compare our solution to existing architectures of the Æthereal family [13, 12, 31] of NoCs that provide end-to-end communication ports in hardware. We address the resource-efficiency by designing our NoC, and mainly the network interface (NI) of our NoC, only to provide communication primitives to the processor and not complete solutions. We use these communication primitives for designing and implementing time-predictable core-to-core communication in an effective and flexible manner.

We have let the three following observations guide our work. First, there are multiple inter-core communication styles that have different requirements to the lower level communication functions. Some communication styles are not coherent with the traditional streaming communications that are implemented by many NoCs. To support as many communication styles as possible, we need a flexible hardware platform with functional primitives that are independent of the higher level semantics.

Second, the problem of creating a TDM schedule can be modeled as a multi-commodity integer flow problem, which is known to be NP-complete [8]. Therefore, for most cases it is not possible to find an optimal TDM schedule, but we can optimize the solution by increasing the flexibility of the TDM schedule, such that the schedule fits the requirements as precise as possible.

Third, real-time applications can have multiple modes of operation, where each mode executes a set of tasks that may or may not overlap with the set of tasks of the other modes. Each set of tasks may have different communication requirements and the union of the communication requirements of all modes may not be supported in a single configuration of the GS channels in the NoC. In case the communication requirements of all modes are not supported in a single NoC configuration, the NoC must support run-time reconfiguration of the GS communication channels.

1.2 Contributions and Organization

This section outlines the contributions of the author, based on the previous observations, in three research areas: Reconfigurable time-division multiplexing network-on-chip, static time-division multiplexing scheduling and time-predictable inter-core communication. For each area we state which publications contribute in this area and how they tie together.

- **Reconfigurable time-division multiplexing network-on-chip**
  Paper A – *An area-efficient TDM NoC supporting reconfiguration for mode changes*

This paper outlines a new and area-efficient NI for the Argo TDM NoC [19]. The architecture of this new NI is a redesign compared to the previous version of the NI. This redesign increases the flexibility of the hardware and adds hardware support for reconfiguring the virtual circuits (VCs). The method of reconfiguring VCs is faster than similar NoCs that execute a reconfiguration incrementally, by tearing down and setting up individual VCs.

Paper B – *A resource-efficient network interface supporting low latency reconfiguration of virtual circuits in time-division multiplexing networks-on-chip*
This paper presents a more detailed description and evaluation of the NI that was outlined in **Paper A**. The paper shows how a compact schedule representation reduces the memory requirements and how variable-length network packets reduces the length of the TDM schedules. The new NI also adds interrupt packets that allow cores to send interrupts to other cores.

- **Static time-division multiplexing scheduling**
  Paper C – *A metaheuristic scheduler for time division multiplexed networks-on-chip*
  This paper presents a metaheuristic TDM scheduler that can generate TDM schedules for our TDM NoC. The TDM schedules enforce the bandwidth of the VCs that an application requires. This scheduler supports the new features of reconfiguration and variable-length packets that we introduced in **Paper A** and **Paper B**. The paper also investigates how we can reduce the period of long TDM schedules, by normalizing the bandwidth requirements.

- **Time-predictable inter-core communication**
  Paper D – *Message passing on a time-predictable multicore processor*
  This paper presents the time-predictable design and implementation of message passing on our multicore platform using the NoC for inter-core communication. The message passing is implemented as a circular buffer, where the receiver needs to acknowledge the received buffer elements. The paper investigates the timing analysis of the implemented message passing functions and of the end-to-end latency of sending a message.

  Paper E – *State-based communication on time-predictable multicore processors*
  This paper studies five time-predictable algorithms for state-based communication. The five algorithms are implemented on our multicore platform using the NoC for inter-core communication. The paper analyses the timing of the read and write functions of the five algorithms and the delay of communicating a state value.

This thesis is organized in 7 chapters. This chapter summarizes the work of the thesis in the areas described above. Chapter 2 reprints **Paper A**. Chapter 3 reprints **Paper B**. Chapter 4 reprints **Paper C**. Chapter 5 reprints **Paper D**. Chapter 6 reprints **Paper E**. Chapter 7 concludes the thesis.

1.3 Reconfigurable Time-division Multiplexing Network-on-Chip

This section first describes how TDM can be used in NoCs. Second, the section introduces the Argo NoC architecture [30, 18, 19] (Argo 1.0) that the work of this thesis builds on. The Argo 1.0 implements TDM very efficiently, but lacks a number of important features to support real-time applications. Third, the section outlines the new generation of the Argo NoC architecture (Argo 2.0) that is presented in **Paper A** and **Paper B**.

1.3.1 Time-Division Multiplexing in Networks-on-Chip

Examples of previous published NoCs that support TDM for routing and static arbitration are Nostrum [25] and the Æthereal family [12, 14, 31]. In a pure TDM NoC that does not support best effort traffic, there is no need for buffers in the routers and no logic to perform dynamic arbitration. This results in simple router hardware and a low network traversal latency of network packets. TDM is not work-conserving and a network packet waits in the source NI until there is an allocated time slot towards its destination. The waiting time, until an allocated time slot, can be much larger than the network traversal time of the packet. Nevertheless, Puffitsch et al. [27] show that the latency guarantees of TDM are generally lower than the latency guarantees of network calculus.

The arbitration in TDM NoCs is implemented by controlling the injection time and the route of network packets with a static TDM schedule. A packet is sent in its statically allocated time slot on a statically allocated route through the network of routers.

Figure 1.1 shows an example of what a TDM schedule is. In the example, the communication requirements in one TDM period are for communication channel C_1, from core p_0 to core p_1, one packet of three time slots and for communication channel C_4, from core p_0 to core p_5, two packets of two time slots. The packets of C_1 can only travel on the shortest path marked as c_1 in the figure and the packets for C_4 can travel on any of the three shortest paths marked as c_4', c_4'', or c_4''' in the figure. A schedule of outgoing packets from core p_0 is shown at the top in Figure 1.1, where C_4 is routed in c_4' and c_4''.

TDM NoCs implement static routing using source routing or distributed routing. Source routing stores the routing information in the NIs and places the route that a packet shall take through the network in the header of an outgoing

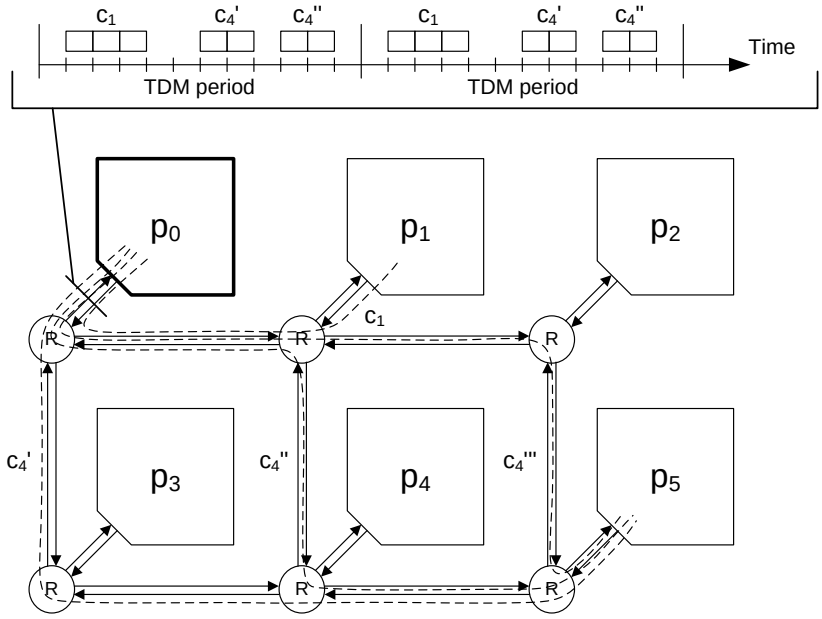


Figure 1.1: A multicore platform with a NoC and an example of a TDM schedule for core p_0 with two outgoing communication channels c_1 and c_4. The marked route c_4''' of c_4 is not used.

packet. An advantage of source routing is that the routers have no local state and therefore do not need to be configured or reset. Distributed routing stores the routing information in the router. Incoming packets are routed according to the schedule stored locally in the router and not according to a packet header. One advantage of distributed routing is that a packet does not need to carry the route, therefore the overhead of transmitting the route with every packet is reduced and the bandwidth is increased. Another advantage is that the length of the route is not limited by the number of bits in a packet header.

1.3.2 Argo 1.0

The Argo 1.0 NoC is a source routed NoC consisting of routers, links, and NIs. Argo 1.0 uses TDM and static scheduling as arbitration of packets to guarantee bandwidth and latency through GS channels. The Argo 1.0 NI implements data transfers from the local scratchpad memory (SPM) of the initiating core to the local SPM of a remote core.

The novelty of the Argo 1.0 NI is that it integrates the direct memory access (DMA) controllers of the local processor into the NI, making the TDM schedule

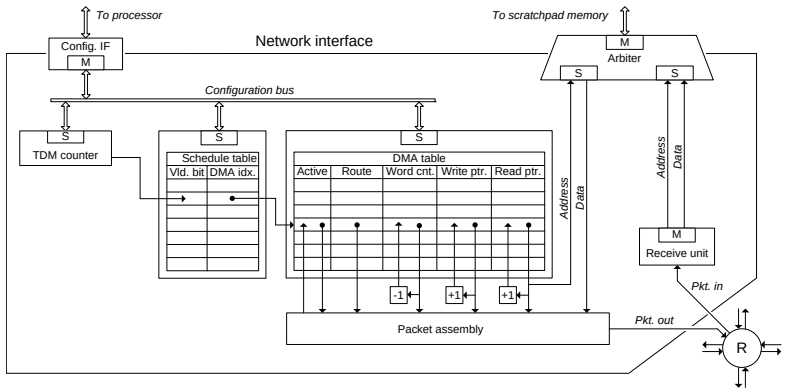


Figure 1.2: The microarchitecture of Argo 1.0.

drive the transmission of packets. Figure 1.2 shows the architecture of Argo 1.0. In Argo 1.0, the number of clock cycles in one time slot is three. Therefore, the TDM counter increments one every three clock cycles and indexes into the schedule table that stores the TDM schedule. A new valid slot from the schedule table activates the DMA controller that is specified in the TDM schedule. The selected DMA table entry dictates the route of the network packets that it sends. When a packet is sent, the fields of the DMA entry are updated accordingly. Incoming packets are written directly into the SPM at the address that they carry in the header, after they have been completely received.

Argo 1.0 has a number of shortcomings that prevent the software from efficiently using the hardware. We outline these shortcomings in the following list. The first four shortcomings possesses research challenges and the last three represent missing features that are solved by engineering work.

Reconfiguration Argo 1.0 lacks the hardware resources to reconfigure the length of the TDM schedule and the TDM schedule itself without resetting all routers and NIs. In Argo 1.0 the length of the TDM schedule cannot be changed during run-time, because the architecture lacks a synchronized way of changing the period length across all cores in the network.

Schedule storage To support large platforms, the number of entries in the slot table and in the DMA table of each Argo 1.0 NI increases. In the slot table, all time slots of a schedule are represented regardless of whether the slots are active or not.

Header overhead The length of packets in Argo 1.0 is fixed to three words, which is one header word and two data words. The header overhead of 33% can be prohibitive in case of large data transfers.

Clock domain crossing through SPM If the processors attached to the Argo 1.0 NoC are operated at frequencies different from the NoC clock frequency, the local SPM of each processor is used to bridge between the global clock domain of the NoC and the local clock domain of each processor. In case the NI of the NoC writes to an address in the SPM at the same time as the processor reads from the same address, there is a risk of propagating metastability into the processor pipeline. Argo 1.0 lacks primitives to prevent the processor from reading an address that is simultaneously written from the NoC clock domain.

Core-to-core interrupts Real-life applications often run on top of an operating systems. A multicore operating system requires the capabilities of interrupting other cores in the platform for several reasons, such as starting and stopping tasks that run on other cores. Therefore, a multicore platform needs to implement core-to-core interrupts to efficiently support a multicore operating system.

Pull communication Argo 1.0 only supports push communication, which is also referred to as a remote write operation. Argo 1.0 lacks the possibility of supporting pull communication, also referred to as a remote read operation. In systems where some nodes in the platform are hardware accelerators or memories, pull communication is needed to read from the memories or get the results from accelerators.

Memory allocation 64-bit granularity In Argo 1.0, the application needs to allocate memory in the SPM in a multiple of 8 bytes and aligned to 8 byte boundaries, because this is what the network packets send. This is due to the fact that sending and receiving a 32-bit word in every clock cycle from the same SPM port requires the port to be 64 bits wide. In a small embedded system this might waste valuable memory.

1.3.3 Argo 2.0

The Argo 2.0 builds on the Argo 1.0 concept of integrating DMA controllers into the NI, using the TDM schedule to activate the DMA controller that is allowed to send in the given time slot. In this section, we first present the new architecture of Argo 2.0 and then in the following subsections we describe how the shortcomings of Argo 1.0 where addressed. The major contributions of Argo 2.0 is the addition of reconfiguration capabilities, addressing the other shortcomings, and a more flexible implementation of the network interface.

1.3.3.1 Architecture

Argo 2.0 supports three packet formats: data packets, configuration packets and interrupt packets. Data packets are used to move a block of data from the local

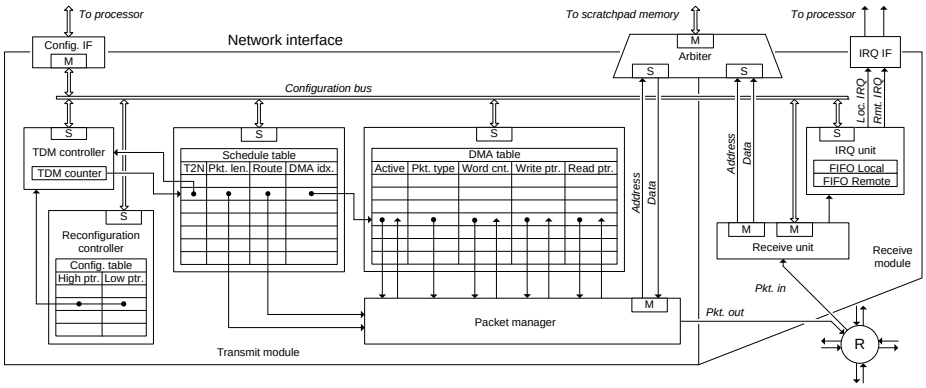


Figure 1.3: The architecture of the second generation Argo NoC.

SPM to the SPM of a remote core. Configuration packets are similar to data packets, but they move a block of data from the local SPM into the tables of the NI in a remote core. Interrupt packets are used to send an interrupt from one core to a remote core. Each packet format consist of a 32-bit header followed by a number of 32-bit payloads. The header consist of a 2-bit packet type, a 14-bit address and a 16-bit route.

The NI architecture of Argo 2.0, as shown in Figure 1.3, is split in two modules: one for transmitting packets and one for receiving packets. The transmit module synchronizes the activation of DMA controllers with the TDM schedule. The TDM controller keeps track of the current time slot and which schedule entry to readout next. This schedule entry contains the index into the DMA table. The packet manager collects the information from the schedule table, the DMA table, and the data from the SPM. From the collected information and the data, the packet manager assembles the packet, updates the entry in the DMA table, and sends it out to the router.

The receive module handles the incoming packets, by either writing the payload to the local SPM or into one of the tables in the NI through the configuration bus. In the case that a received packet is marked as the last packet of a data transfer, the address of the last word of that packet is stored in the IRQ local first in first out (FIFO). In the case of an interrupt packet, the payload is written to the address in the local SPM that is specified in the interrupt header. Furthermore, the address of the interrupt header is written into the IRQ unit, where it is stored in the IRQ Remote FIFO. Storing the header address in one of the IRQ FIFOs triggers either a remote interrupt or a local interrupt in the processor.

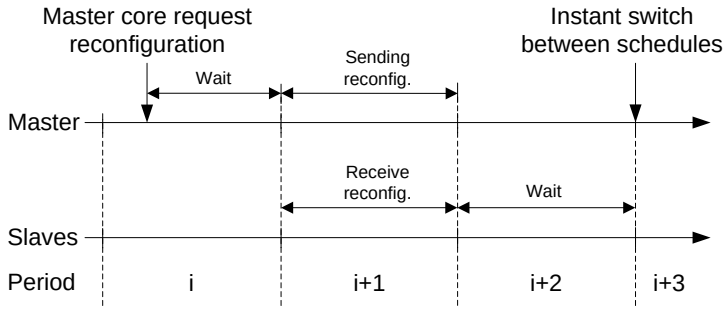


Figure 1.4: A timing diagram of the reconfiguration method.

1.3.3.2 Reconfiguration

Our approach of reconfiguring the TDM schedule of the NoC is to instantaneously switch from one TDM schedule to another across the whole platform. The switch to a new schedule is aligned to a TDM period boundary. This approach requires that the TDM schedule table can store the old and the new schedule at the same time. A schedule is represented by a number of entries in the schedule table and two pointers to the first and to the last entry of the schedule. These two pointers are stored in the configuration table of the transmit module.

Figure 1.4 shows a timing diagram of our reconfiguration method. To reconfigure the NoC, the master core writes the index of the schedule that the NoC should switch to into the reconfiguration controller of the NI. On the next schedule period boundary, the NI of the master core sends a packet to each reconfiguration controller in the slave NIs. These configuration packets contain the index of the configuration that all NIs should switch to and the TDM period number when they should execute the switch. Three TDM periods after the master core requested its NI to change the configuration of the TDM schedule, the TDM schedule is switched to the new schedule synchronously across the entire platform. This reconfiguration approach requires that the current TDM schedule contains GS channels from the master to all slave cores. These GS channels can be allocated exclusively to reconfiguration, if it is necessary to avoid changing the timing of active DMA transfers from the master.

In the Æthereal family of NoCs the reconfiguration is performed by incrementally tearing down and setting up virtual circuits in the active TDM schedule. That approach suffers from fragmentation in the TDM schedule and a lengthy transition of the schedule table from one schedule to another.

In Argo 2.0 there are two scenarios in which a reconfiguration can be carried out: one scenario is when the new TDM schedule is already in the schedule table, the other scenario is when the new TDM schedule is not in the schedule

Table 1.1: The worst-case reconfiguration time (WCRT) for the selection of benchmarks.

Benchmark	WCRT
FFT-1024	267
Fpppp	285
RS-dec	279
RS-enc	261
H264-720p	279
Robot	381
Sparse	135
All2all	270

Table 1.2: The worst-case schedule transmission time for the selection of benchmarks.

C_{curr} \ C_{new}	FFT-1024	Fpppp	RS-dec	RS-enc	H264-720p	Robot	Sparse	All2-all
FFT-1024	–	2010	1418	1270	1341	1560	1122	2229
Fpppp	2577	–	1814	1624	1716	2004	579	2862
RS-dec	2091	2088	–	1318	1398	1626	1164	2316
RS-enc	1983	1980	1396	–	1326	1548	1104	2196
H264-720p	2115	2112	1494	1338	–	1644	1176	2355
Robot	3435	3438	2422	2174	2292	–	1914	3822
Sparse	822	549	579	519	546	639	–	912
All2all	2034	2037	1431	1281	1365	1587	1137	–

table and needs to be transmitted through the NoC. We call the time to carry out a reconfiguration for the reconfiguration time, and the time to transmit a new schedule through the NoC for the schedule transmission time. In the first scenario, the worst-case reconfiguration time is three times the length of the TDM period of the currently executing schedule. In the second scenario, the schedule transmission time is added to the reconfiguration time of the first scenario.

We use the benchmarks of the MCSL benchmark suite [22] to evaluate the reconfiguration times of our new architecture. Table 1.1 shows the worst-case reconfiguration time for each schedule being the current schedule. Table 1.2 show the worst-case schedule transmission time of each schedule (columns) with respect to the currently executing schedule (rows). It is clear that the additional worst-case schedule transmission time is much higher than the worst-case reconfiguration time. To reduce the occurrences of transmitting a schedule through

Table 1.3: The schedule compression ratio of the Argo 2.0 schedule over the Argo 1.0 schedule.

Benchmark	Argo 1.0 (entries)	Argo 2.0 (entrites)	Reduction (%)
FFT-1024	21	15	28.6
Fpppp	40	16	60.0
RS-dec	30	8	73.3
RS-enc	28	6	78.6
H264-720p	30	7	76.7
Robot	57	10	82.5
Sparse	9	4	55.6
All2all	18	16	11.1

the NoC it is a great advantage that as many schedules as possible fit in the schedule table.

1.3.3.3 Schedule Storage and Header Overhead

We can make the representation of a schedule more dense by only representing active time slots. An active time slot is a slot that represents the event of transmitting a packet header. We avoid representing empty slots in the schedule table and slots that represent the payload cycles, by storing the time between packet headers in the T2N field and the length of a packet in the pkt. len. field of each entry, respectively.

Encoding the length of a packet in the pkt. len. field allows us to send packets of variable length. Longer packets reduces the overhead of transmitting headers and increases the time between packet headers. Table 1.3 shows the compression ratio between Argo 2.0 and Argo 1.0 for each benchmark schedule. We observe that the number of entries for most of the benchmarks are reduced by more than 50%.

Our compact schedule representation can also be used with the incremental approach of reconfiguration that is used by the Æthereal family of NoCs. In the compact schedule representation, the time to transmit a packet is based on the time that the previous packet was sent plus the offset in the T2N field. Thus, changing the T2N field in an entry in a schedule table will change the TDM period of the active schedule. Therefore, the user can execute incremental updates of the schedule, but must not change any of the T2N fields. This may sound restricting, but by setting all T2N fields to three, the NI supports the same class of schedules as the Æthereal family of NoCs. In this way, the user can atomically update an entry in the schedule table without affecting other entries.

1.3.3.4 Clock Domain Crossing Through SPM

When a processor continuously reads a memory address in the SPM to detect whether it has received data through the NoC, the problem of reading while writing can occur. The effect of this problem can be eliminated by creating a mechanism where the reader can detect if it received the expected data before it reads from that address in the SPM.

Our mechanism to prevent the problem is to trigger an interrupt when the last data packet of a DMA transfer arrives in the NI. The last data packet triggers an interrupt after the NI has written the last data of a DMA transfer into the destination SPM. To prevent an incoming interrupt from interfering with an executing task, the processor should mask the interrupt when it is not trying to receive a message. This feature allows the sender and receiver to coordinate at the data block level and avoid metastability. To identify the VC that caused the interrupt, we store the address of the last data word written into the SPM in a FIFO queue.

1.3.3.5 Addressing the Remaining Shortcomings

We support core-to-core interrupt capabilities by adding an interrupt packet format. The interrupt packet contains 32 bits of data that can be used as an interrupt identifier, a function pointer or some kind of command. The 32-bit data word is stored at the address specified in the interrupt packet header and that address is pushed into the IRQ FIFO for remote interrupts.

We implement pull communication or remote reads by sending a configuration packet that writes into the DMA table of the NI from where we want to read. The configuration packet sets up a DMA transfer of the data block that we want to read from SPM of the remote NI. The worst-case latency of a remote read consist of the worst-case of two remote writes. One remote write with 64-bit payload to setup the DMA transfer in the remote core, and one remote write transferring the desired data from the remote core to the local core.

In Argo 2.0, we use an SPM from which we can read or write a 32-bit or a 64-bit word in each clock cycle aligned to a 32-bit address. With this SPM we can send and receive packets with any number of 32-bit payload words, thereby reducing the minimum amount of allocatable memory to 32 bits.

1.4 Static Scheduling in Time-Division Multiplexing Networks-on-Chip

This section introduces: the application and platform model that our TDM scheduler uses, the scheduling algorithm that schedules the packets, and two features of the scheduler that improve the flexibility and performance of the

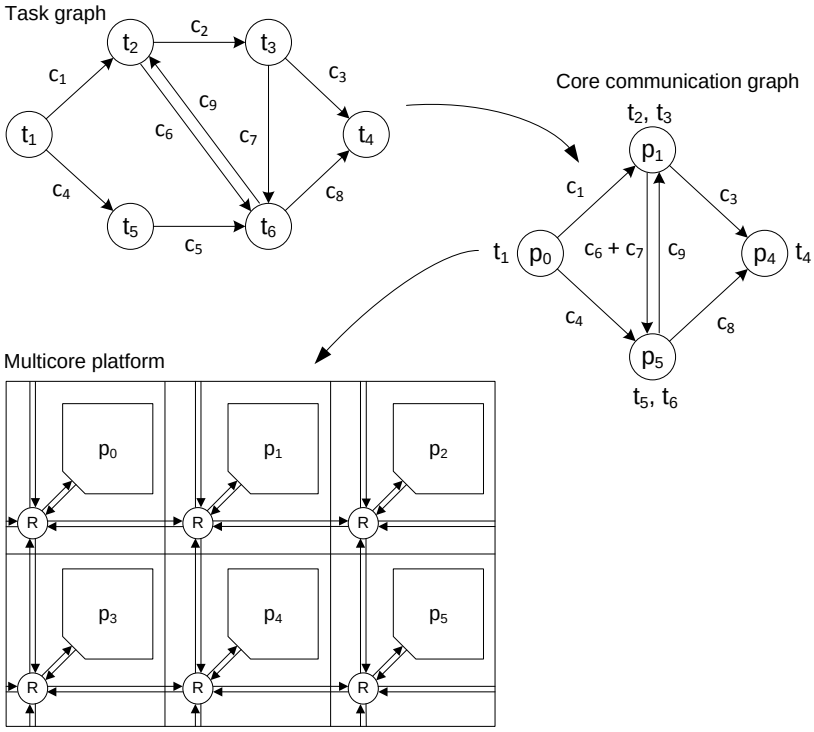


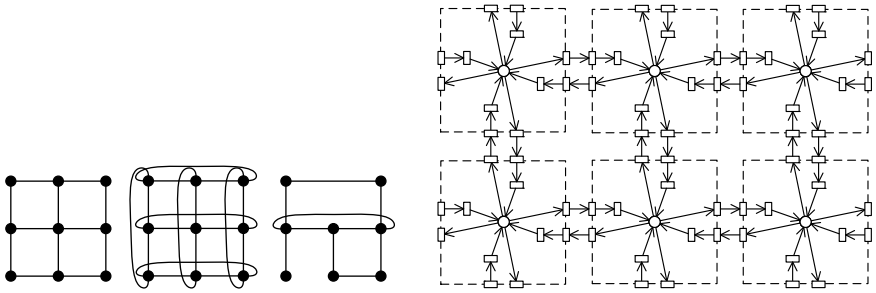
Figure 1.5: The mapping of an application onto a multi-core platform from a task graph of an application through a core communication graph to a multi-core platform.

generated schedules. This section outlines **Paper C**, but work performed after the publication of **Paper C** is also presented here. In **Paper C** the numbers of the TDM period are in time slots, where a single time slot is 3 clock cycles. Whereas, the numbers in the other papers are in clock cycles.

1.4.1 Application and Platform Model

A real-time application that runs on a multicore platform has requirements to the communication bandwidth between the communicating cores. In a NoC for real-time systems these bandwidth requirements are setup as unidirectional GS communication channels, with guarantees on the bandwidth and latency.

We assume that an application can be represented as a task graph, as shown in Figure 1.5, where the nodes are the tasks and the edges are the unidirectional bandwidth requirements between the tasks. Furthermore, we assume that tasks



(a) Examples of topologies that are supported by our scheduler. The links can be unidirectional or bidirectional and each node is connected to one IP.

(b) An example of the abstract RTL view, assumed by our scheduler, of a NoC with routers of three pipeline stages. The dashed lines encapsulate a router.

Figure 1.6: The supported platform model of our scheduler at the node level and at the register transfer level (RTL).

are statically mapped to processing cores. We call this mapping a core communication graph. The task of the scheduler is to map the bandwidth requirements of the core communication graph onto the NoC with a TDM schedule.

The bandwidth requirements of an application are in MB/s. The scheduler needs to know how many NoC packets that it needs to schedule within one TDM period for each channel. The scheduler needs the bandwidth requirements in MB/s converted to bandwidth requirements in number of NoC packets per TDM period.

We model our platform as a 2 dimensional array of nodes with possible connections between the edges to allow topologies with rings, such as a torus. Furthermore, we do not require that all nodes are present in the 2 dimensional array, to allow the scheduler to support irregular NoC topologies. Figure 1.6a shows three examples of topologies that are supported by our scheduler: a mesh network, a torus network, and an irregular custom network. Each router can connect to one processing core or an intellectual property (IP) and to four other routers. We assume that all routers have the same pipeline depth, where no packets traverse a router faster than others, and that individual links can have arbitrary pipeline depths. These assumptions can be relaxed by assuming that the pipeline depth of routers is one and assigning the remaining pipeline stages of each router to its incoming links, so that any non-zero pipeline depth of routers is possible.

Figure 1.6b shows an example of an abstract register transfer level (RTL) view of a network with routers of three pipeline stages and link of zero pipeline stages. The crossbar in the routers are the only branching points. The number of pipeline stages between each router can be configured, but packets are only

routed on a path between the source and destination, with the minimum number of pipeline registers. This RTL model is generic and it only assumes that routers and links are bufferless, but makes no assumptions of the router and link pipeline depth. We expect that this abstract model will fit most NoCs that use TDM. Scheduling at the level of single pipeline stages allows us to schedule packets with arbitrary length on platforms where routers and links have arbitrary pipeline depth. On the other hand, if this low level scheduling is not needed and the length of the packets is a multiple of the pipeline depth of the routers, the user can model the pipeline depth of the routers as a single pipeline stage in the network model.

1.4.2 The Scheduling Algorithm

Besides fulfilling the bandwidth requirements, a valid schedule is required to: ensure in-order arrival of packets, guarantee that no packets collide, and prevent deadlocks from occurring. Our scheduler ensures that all packets of a communication channel arrives at their destination in the same order as they were sent, by restricting packets only to travel on shortest paths. Our scheduler allocates a sequence of consecutive links from the source to the sink, only if the links are available, so there can be no collisions of packets in the generated schedule. Freedom of deadlocks is trivially obtained, because there is no flow control in the network, thus no packets can block other packets.

Our scheduling algorithm has two phases. Phase one is an all-pair shortest path search that finds all possible packet routes, for any combination of source and sink nodes. Phase two is the allocation of routing resources to communication channels. The routing resources are links and pipeline registers that are divided in time slots. In one time slot, one word of a NoC packet can traverse one link and the following register. Our algorithm allocates all the required communication channels in a graph that is a time expansion of the platform. The graph is expanded in time to as many time slots as the bandwidth requirements dictate. The depth of the pipelines between routers are modeled by connecting the outgoing edges to the neighboring routers in as many time slots later as the pipeline to the router dictates. Therefore, the complexity of the routing problem does not changes with the pipeline depth in the platform model.

The allocation of routing resources is done in an iterative approach. First, a greedy algorithm generates a valid solution and then that solution is optimized using a metaheuristic algorithm until a termination criterion is satisfied, in this case time. The scheduler implements two metaheuristic algorithms: greedy randomized adaptive search procedure (GRASP) [9] and adaptive large neighborhood search (ALNS) [28]. The user can select which metaheuristic to use and for how long to run it or just run the greedy algorithm once.

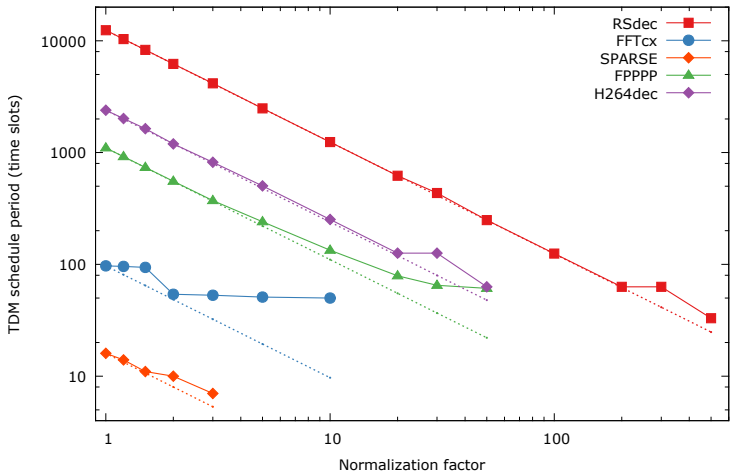


Figure 1.7: The TDM period length of five of the MCSL benchmarks, as a function of the normalization factor.

1.4.3 Bandwidth Normalization

A real application specifies its bandwidth requirements in MB/s, while a TDM schedule is generated with a number of time slots in a TDM period. So the scheduler needs the requirements in MB/s converted to a number of network packets per TDM period. The requirements of each communication channel in the core communication graph can be converted to a number of network packets by normalizing with a common factor and assigning at least one packet to a communication channel.

The common factor is bound by the highest and lowest bandwidth requirements of the core communication graph. Normalizing to the lowest bandwidth requirement results in a communication pattern where the bandwidth ratio of the highest and lowest bandwidth requirements are intact. This normalization can lead to a high number of network packets within one TDM period if the bandwidth ratio, between the lowest and highest bandwidth requirements, is high.

Normalizing to the highest bandwidth requirement results in a communication pattern where all bandwidth requirements are one, the least possible. This normalization can lead to a TDM period that does not provide the desired bandwidth.

To satisfying the bandwidth requirements with as few TDM slots as possible, we investigate how the guaranteed bandwidth varies as we increase the normalization factor from the lowest bandwidth requirement. Figure 1.7 shows the TDM period length of five benchmark applications as a function of the nor-

Benchmark	Fixed-len. pkt.	Var.-len. pkt.	
	(cc)	(cc)	(%)
FFT-1024	78	74	5
Fpppp	120	95	21
RS-dec	92	77	16
RS-enc	86	73	15
H264-720p	92	78	15
Robot	171	127	26
Sparse	30	30	0
All2all	75	75	0

Table 1.4: Reduction (%) of TDM period in clock cycles (cc) due to variable-length packets.

malization factor. If the TDM period scales down inversely proportional to the normalization factor, then the highest bandwidth requirements of a communication channel are maintained. In Figure 1.7, the dashed lines represent the linear scale down where the bandwidth is maintained. Increasing the normalization factor while the TDM period scales down along the dashed line does not reduce the bandwidth. The user can allow a decrease in the provided bandwidth compared to the dashed line in case the TDM period is too long to be implemented in a practical system.

1.4.4 Variable-Length Packets

As mentioned in Section 1.3 the new generation of the Argo NoC supports variable-length packets, by using the compact schedule representation. Longer packets reduce the overhead of transmitting packet headers, by sending fewer and longer packets. **Paper B** shows how much the scheduler can reduce the TDM periods of the MCSL benchmarks by allowing variable-length packets. Table 1.4 shows the TDM period reductions in clock cycles due to variable-length packets. In percent, the TDM period reduction is equal to the increase in bandwidth.

The largest TDM period reductions occur for the Fpppp and the Robot benchmarks. These benchmarks have a few cores that have high bandwidth requirements and the TDM period reduction is due to the reduction in header overhead of these high bandwidth requirements.

1.5 Time-Predictable Inter-Core Communication

This section gives an overview of the topics from **Paper D** and **Paper E**, by presenting the time-predictable inter-core communication that executes on the multicore platform using the NoC to transfer data between cores.

1.5.1 Communication in Real-time Systems

Communication between the processes of a parallel application can in the source code be specified implicitly or explicitly. Implicit communication is when processes exchange data in a way that is not directly visible from the source code. This could be through a shared data structure, where the processes communicate by reading and writing to the same memory address. Explicit communication is when processes exchange data in a way that is visible from the source code. This could be through a unidirectional communication channel that is explicitly accessed in the source code by a send or receive primitive. Examples of languages or programming models that use explicit communication are Kahn process networks [16], communicating sequential processes [15], and synchronous dataflow [21].

Explicit communication is the required communication style between partitions, also referred to as processes, in the ARINC (Aeronautical Radio, Incorporated) [2] 653 specification. Explicit communication has different semantics. The ARINC standard specifies two types of explicit communication between partitions: queuing communication and sampling communication. Queuing communication is as the name suggests communication through a queue, meaning that the consumer needs to consume all messages that are produced by the producer. This type of communication is generally referred to as message passing, which we will use throughout the thesis. Sampling communication means that the producer and consumer communicates through a single sample that is updated atomically, such that the consumer does not need to read all values that are produced, but only reads the newest one. The sampling communication values are also referred to as state messages [20], so we use the term state-based communication.

A time-predictable platform must provide end-to-end worst-case latencies of messages to an application running on the platform. As we are concerned with domain-specific platforms that support a wide range of applications, we require the platform to provide a software interface with communication primitives that can be statically analyzed.

1.5.2 Message Passing

If two processes communicate through a message queue, the consumer needs to consume all messages that are produced. Therefore, the maximum frequency

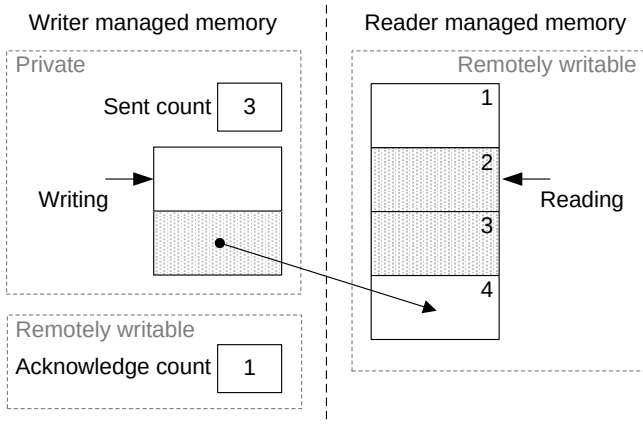


Figure 1.8: The memory layout of a message queue with four elements.

at which the producer produces messages must be lower than the worst-case frequency, i.e., the minimum frequency, at which the consumer consumes messages.

To implement a message queue, we need to move messages from the local SPM of the sender to the local SPM of the receiver. These messages can be moved through the NoC from the sender to the receiver in two ways: the sender pushes the messages to the receiver or the receiver pulls the messages from the sender.

We have implemented the message queue as a circular buffer in the receiver with the sender pushing messages into the circular buffer. The receiver polls to check whether it has received a message. To avoid unnecessary copying of data, the receiver operates on the head of the message queue. When the receiver is done with the current head of the message queue, the receiver sends an acknowledgment to the sender, indicating that the buffer element is again free. The acknowledgment count is the total number of messages that the receiver has acknowledged since the queue was set up. The sender compares the number of messages that the receiver acknowledged, with the number of messages that the sender sent since the queue was set up. This difference is the fill level of the message queue in the receiver.

To overlap communication and computation in the sender, we implement double buffering at the sender side, such that the sender can write data of a new message while the NI is transmitting the previous message. Figure 1.8 shows the memory layout of the message queue in a situation where the producer is writing a new message, while the NI of the producer is transferring the previous message. On the other side, the consumer has acknowledged a single message and is reading the new head of the queue.

With the new features of Argo 2.0, the receiver receives an interrupt when a transfer has completed. Therefore, the receiver can choose to disable interrupts and then poll the pending interrupt register to check if a message has arrived. When the receiver detects the interrupt, the receiver can read out the address of the last written word of the transfer to verify that the interrupt came from the correct VC. This method avoids reading memory locations that might be written at the same time by an incoming network packet.

The worst-case execution time (WCET) of sending or receiving a message, as seen from the processing core, is independent of the message size, because the copying or transferring of the message is done by the NI in parallel with execution of the processing core. The transfer time of the message through the NoC depends on the bandwidth of the communication channel that is allocated towards the destination.

1.5.3 State-Based Communication

If two processes communicate through state-based communication, the reader must always read the newest completely written state value. A write or read of the newest state value must be executed atomically, such that the reader does not see a mix of an old and a new value.

We have implemented five algorithms for state-based communication and evaluated their WCET and their worst-case communication delay (WCCD). The first algorithm has a single shared buffer that is protected by a lock. The writer and reader acquire the lock to copy the full state value to or from the buffer. The critical section of the writer contains the transfer of the whole state value through the NoC and the critical section of the reader contains the copying of the whole state value from the local SPM.

The second and third algorithm use two shared buffers protected by a lock. In the second algorithm, the writer acquires the lock to update which buffer contains the newest state value and the reader acquires the lock to read the entire state value. The critical section of the writer only contains the updating of which buffer contains the newest state value and the critical section of the reader still contains copying the whole state value.

In the third algorithm, the writer acquires the lock to transfer the entire state value to the reader and the reader acquires the lock to get the buffer index with the newest state value and to update which buffer it is reading. The critical section of the writer contains the transfer of the whole state value through the NoC and the critical section of the reader contains the updating of which state value it is reading.

The fourth algorithm use three shared buffers protected by a lock. The writer acquires the lock to update which buffer contains the newest state value and the reader acquires the lock to update which buffer it is reading. The critical section of the writer only contains the updating of which state value

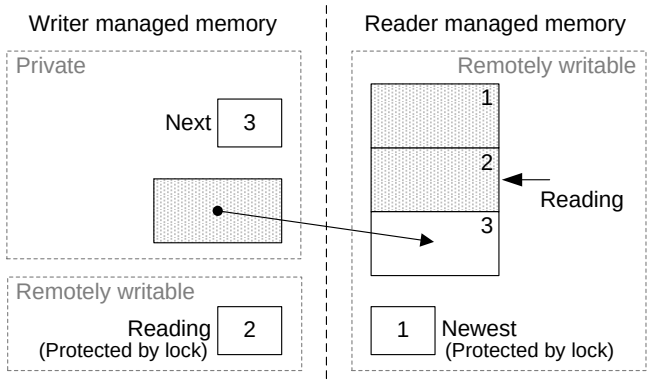


Figure 1.9: The memory layout of state-based communication.

is the newest and the critical section of the reader only contains the updating of which state value it is reading. Figure 1.9 shows the memory layout of the protocol with three shared buffers and a lock.

The fifth algorithm uses a message queue, where the reader dequeues all the elements and only keeps the newest state value. We used the message queue that we implemented for the message passing. This protocol requires that the number of elements in the message queue is larger than the ratio of the writing frequency over the reading frequency, this is to ensure that there are always free buffers in the queue.

For the four algorithms that use a lock on the Argo 2.0 hardware, the reader does not need a local interrupt when a data transfer has completed, because the mutual exclusion is enforced by the lock. Therefore, the writer setup data transfers via the NoC that do not trigger a local interrupt at the reader. For the algorithm that uses the message passing queue, the Argo 2.0 features are used as described in the previous section.

The timing analysis of the five algorithms show that they each have their advantages and disadvantages and out perform the other algorithms depending on which performance metric is more valued. Most remarkably, the highest WCET of the write and read functions do not result in the longest WCCD from one core to another.

Bibliography

[1] J. Andersson, J. Gaisler, and R. Weigand. Next generation multipurpose microprocessor. *DASIA, Budapest, Hungary*, 2010.

[2] ARINC 653. Avionics application software standard snterface – Part 1: Required services, 2010.

[3] G. Bartlett. Qoriq p4080 communications processor design in 45nm soi. In *2009 IEEE International Conference on IC Design and Technology*, pages viii–viii, May 2009. doi: 10.1109/ICICDT.2009.5166246.

[4] T. Bjerregaard and S. Mahadevan. A survey of research and practices of network-on-chip. *ACM Comput. Surv.*, 38(1), June 2006. ISSN 0360-0300. doi: http://doi.acm.org/10.1145/1132952.1132953.

[5] T. Bjerregaard and J. Sparso. Implementation of guaranteed services in the mango clockless network-on-chip. *IEE Proceedings - Computers and Digital Techniques*, 153(4):217–229, July 2006. ISSN 1350-2387. doi: 10.1049/ip-cdt:20050067.

[6] B. D. de Dinechin, P. G. de Massas, G. Lager, C. Léger, B. Orgogozo, J. Reybert, and T. Strudel. A distributed run-time environment for the kalray mppa®-256 integrated manycore processor. *Procedia Computer Science*, 18:1654–1663, 2013.

[7] B. D. de Dinechin, Y. Durand, D. van Amstel, and A. Ghiti. Guaranteed services of the noc of a manycore processor. In *Proceedings of the 2014 International Workshop on Network on Chip Architectures*, NoCArc '14, pages 11–16, New York, NY, USA, 2014. ACM. ISBN 978-1-4503-3064-0. doi: 10.1145/2685342.2685344.

[8] S. Even, A. Itai, and A. Shamir. On the complexity of time table and multi-commodity flow problems. In *Foundations of Computer Science, 1975., 16th Annual Symposium on*, pages 184 –193, oct. 1975.

[9] T. A. Feo and M. G. Resende. Greedy randomized adaptive search procedures. *Journal of Global Optimization*, 6(2):109–133, March 1995. ISSN 0925-5001. doi: 10.1007/BF01096763.

[10] M. Fernández, R. Gioiosa, E. Quiñones, L. Fossati, M. Zulianello, and F. J. Cazorla. Assessing the suitability of the ngmp multi-core processor in the space domain. In *Proceedings of the Tenth ACM International Conference on Embedded Software*, EMSOFT '12, pages 175–184, New York, NY, USA, 2012. ACM. ISBN 978-1-4503-1425-1. doi: 10.1145/2380356.2380389.

[11] D. Geer. Chip makers turn to multicore processors. *Computer*, 38(5):11–13, May 2005. ISSN 0018-9162. doi: 10.1109/MC.2005.160.

[12] K. Goossens and A. Hansson. The aethereal network on chip after ten years: Goals, evolution, lessons, and future. In *Design Automation Conference (DAC), 2010 47th ACM/IEEE*, pages 306–311, 2010.

[13] K. Goossens, J. Dielissen, and A. Radulescu. Aethereal network on chip: concepts, architectures, and implementations. *IEEE Design Test of Computers*, 22(5):414–421, Sept 2005. ISSN 0740-7475. doi: 10.1109/MDT.2005.99.

[14] A. Hansson, M. Subburaman, and K. Goossens. aelite: a flit-synchronous network on chip with composable and predictable services. In *Proceedings of the Conference on Design, Automation and Test in Europe*, DATE '09, pages 250–255, 3001 Leuven, Belgium, Belgium, 2009. European Design and Automation Association. ISBN 978-3-9810801-5-5.

[15] C. A. R. Hoare. Communicating sequential processes. *Communications of the ACM*, 21(8):666–677, AUG 1978. Reprinted in "Distributed Computing: Concepts and Implementations" edited by McEntire, O'Reilly and Larson, IEEE, 1984.

[16] G. Kahn. The semantics of a simple language for parallel programming. In J. L. Rosenfeld, editor, *Information Processing 74: Proceedings of the IFIP Congress 74*, pages 471–475. IFIP, North-Holland Publishing Co., Aug. 1974.

[17] E. Kasapaki. *An Asynchronous Time-Division-Multiplexed Network-on-Chip for Real-Time Systems, Technical University of Denmark (DTU).* PhD thesis, 2015.

[18] E. Kasapaki, J. Sparsø, R. B. Sørensen, and K. Goossens. Router designs for an asynchronous time-division-multiplexed network-on-chip. In *Digital System Design (DSD), 2013 Euromicro Conference on*, pages 319–326. IEEE, 2013.

[19] E. Kasapaki, M. Schoeberl, R. B. Sørensen, C. Müller, K. Goossens, and J. Sparsø. Argo: A real-time network-on-chip architecture with an efficient gals implementation. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, 24(2):479–492, Feb 2016. ISSN 1063-8210. doi: 10.1109/TVLSI.2015.2405614.

[20] H. Kopetz. *Real-Time Systems.* Kluwer Academic, Boston, MA, USA, 1997.

[21] E. A. Lee and D. G. Messerschmitt. Synchronous data flow. *Proceedings of the IEEE*, 75(9):1235–1245, Sept 1987. ISSN 0018-9219. doi: 10.1109/PROC.1987.13876.

[22] W. Liu, J. Xu, X. Wu, Y. Ye, X. Wang, W. Zhang, M. Nikdast, and Z. Wang. A noc traffic suite based on real applications. In *VLSI (ISVLSI), 2011 IEEE Computer Society Annual Symposium on*, pages 66 –71, july 2011.

[23] Z. Lu and A. Jantsch. TDM virtual-circuit configuration for network-on-chip. *IEEE Trans. Very Large Scale Integr. Syst.*, 16:1021–1034, August 2008. ISSN 1063-8210.

[24] D. Melpignano, L. Benini, E. Flamand, B. Jego, T. Lepley, G. Haugou, F. Clermidy, and D. Dutoit. Platform 2012, a many-core computing accelerator for embedded socs: performance evaluation of visual analytics applications. In *Proceedings of the 49th Annual Design Automation Conference*, pages 1137–1142. ACM, 2012.

[25] M. Millberg, E. Nilsson, R. Thid, and A. Jantsch. Guaranteed bandwidth using looped containers in temporally disjoint networks within the nostrum network on chip. In *Design, Automation and Test in Europe Conference and Exhibition, 2004. Proceedings*, volume 2, pages 890 – 895 Vol.2, feb. 2004.

[26] D. A. Patterson and D. R. Ditzel. The case for the reduced instruction set computer. *SIGARCH Comput. Archit. News*, 8(6):25–33, Oct. 1980. ISSN 0163-5964. doi: 10.1145/641914.641917.

[27] W. Puffitsch, R. B. Sørensen, and M. Schoeberl. Time-division multiplexing vs network calculus: A comparison. In *Proceedings of the 23rd International Conference on Real Time and Networks Systems*, RTNS '15, pages 289–296, New York, NY, USA, 2015. ACM. ISBN 978-1-4503-3591-1. doi: 10.1145/2834848.2834868.

[28] S. Ropke and D. Pisinger. An adaptive large neighborhood search heuristic for the pickup and delivery problem with time windows. *Transportation science*, 40(4):455–472, November 2006.

[29] Z. Shi and A. Burns. Real-time communication analysis for on-chip networks with wormhole switching. In *Proceedings of the Second ACM/IEEE International Symposium on Networks-on-Chip*, NOCS '08, pages 161–170, Washington, DC, USA, 2008. IEEE Computer Society. ISBN 978-0-7695-3098-7.

[30] J. Sparsø, E. Kasapaki, and M. Schoeberl. An area-efficient network interface for a TDM-based network-on-chip. In *Design, Automation Test in Europe Conference Exhibition (DATE), 2013*, pages 1044–1047, March 2013. doi: 10.7873/DATE.2013.217.

[31] R. Stefan, A. Molnos, and K. Goossens. daelite: A tdm noc supporting qos, multicast, and fast connection set-up, 2012. ISSN 0018-9340.

[32] S. Tobuschat, P. Axer, R. Ernst, and J. Diemer. Idamc: A noc for mixed criticality systems. In *2013 IEEE 19th International Conference on Embedded and Real-Time Computing Systems and Applications*, pages 149–156, Aug 2013. doi: 10.1109/RTCSA.2013.6732214.

[33] D. Wiklund and D. Liu. Socbus: switched network on chip for hard real time embedded systems. In *Parallel and Distributed Processing Symposium, 2003. Proceedings. International*, pages 8 pp.–, April 2003. doi: 10.1109/IPDPS.2003.1213180.

Chapter 2

An Area-Efficient TDM NOC Supporting Reconfiguration for Mode Changes

This chapter was previously published: © 2016 IEEE. Reprinted, with permission, from Rasmus Bo Sørensen, Luca Pezzarossa, and Jens Sparsø,
"An Area-Efficient TDM NOC Supporting Reconfiguration for Mode Changes",
10th IEEE/ACM International Symposium on Networks-on-Chip (NOCS), 2016.

Abstract

This paper presents an area-efficient time-division-multiplexing (TDM) network-on-chip (NoC) intended for use in a multicore platform for hard real-time systems. In such a platform, a mode change at the application level requires the tear-down and set-up of some virtual circuits without affecting the virtual circuits that persist across the mode change. Our NoC supports such reconfiguration in a very efficient way, using the same resources that are used for transmission of regular data. We evaluate the presented NoC in terms of worst-case reconfiguration time, hardware cost, and maximum operating frequency. The results show that the hardware cost for an FPGA implementation of our architecture is a factor of 2.2 to 3.9 times smaller than other NoCs with reconfiguration functionalities, and that the worst-case time for a reconfiguration is shorter or comparable to those NoCs.

2.1 Introduction

Packet-switched networks-on-chips (NoCs) have become the preferred paradigm for interconnecting the many cores (processors, hardware accelerators, etc.) found in complex application-specific multi-processor systems-on-chip [4, 1].

Hard real-time multicore applications rely on the guaranteed-service (GS) of the NoC, in terms of bandwidth and latency for end-to-end virtual circuits, in order to guarantee correct timing behavior. This class of applications often has multiple modes of operations that they switch between during normal operation. A mode change consists of a change of a subset of the executing software tasks and it can be triggered as part of the normal operation of the system or in response to external events [3, p.340]. To support applications that change between modes, the NoC must be able to reconfigure the GS connections during run-time, since a single configuration of the time-predictable NoC may not support all modes of operation for a given real-time application.

This paper proposes and evaluates a flexible and resource-efficient NoC for use in the hard real-time domain. The proposed NoC extends the existing NI of the Argo NoC [10] to support mode changes. This extension is the main contribution of the paper, and a key feature of the proposed NoC. Our NoC implements virtual end-to-end circuits using static scheduling and time-division multiplexing (TDM). The Æthereal family of NoCs [5, 11] provides similar functionality at a much higher hardware cost.

The key idea of the Argo NI [10] is that the DMA controllers that drive the data transfers are integrated with the TDM scheduling in the NIs. This architecture avoids both *physical* virtual channel buffers in the NIs and credit-based flow control among the NIs that are found in most other NoC designs [2, 8, 11]. Since these resources must be reconfigured as part of a mode change [5, 11], the NI architecture of [10] can be extended to support mode changes using little extra hardware as we show in this paper.

This paper is organized in seven sections. Section 2.2 presents related work. Section 2.3 presents the overall functionality of the network interface (NI) of the presented NoC and Section 2.4 describes its hardware architecture. Section 2.5 presents the controller that manages the reconfiguration process. Section 2.6 evaluates the presented architecture. Section 2.7 concludes the paper.

2.2 Related Work

A multicore platform for hard real-time systems has to provide time-predictable inter-processor communication. In the following we present some TDM NoCs that offer GS connections and support run-time reconfiguration of the GS connections.

The Æthereal family of NoCs [5, 11] uses TDM and static scheduling to provide GS. The original Æthereal NoC supports both GS and BE traffic. The scheduling and routing tables controlling the GS traffic are distributed into the NIs and routers. The data structures that are written into these tables are distributed using BE traffic. The use of BE traffic for reconfiguration may compromise the time-predictability. Observing the high cost of distributed routing and combined support for BE and GS traffic, the aelite NoC [5] supports only GS traffic and source routing. Reconfiguration is performed using GS connections (reserved for this purpose only) from a reconfiguration master. Essentially, our NoC implements similar functionality using fewer hardware resources. The dAElite NoC, focusing on multicast, re-introduces distributed routing and adds a dedicated network with a tree topology to handle reconfiguration.

The Argo NoC [6] supports GS traffic and uses TDM with source routing like the aelite NoC, but Argo does not support the reconfiguration needed to do a mode change. In the Argo NoC, the TDM schedule drives the transmission of data between scratchpad memories (SPM) attached to the processors in each node of the platform. The Argo network interface (NI) contains a direct memory access (DMA) table, where the local processor can setup DMA transfers to remote SPMs. The TDM schedule activates a certain DMA entry when it is allowed to send a packet. The Argo NoC uses TDM schedules generated by the off-line scheduler presented in [9]. We assume that each mode consists of a set of communicating tasks assigned to processors, and that each mode has an associated bandwidth graph, from which the scheduler can generate a schedule that provides the GS requirements of each virtual circuit for each mode of operation.

2.3 Network Interface Functionality

This section describes the key ideas of the reconfiguration feature and outlines the NI architecture, shown in Fig. 2.1.

In our resource-efficient NoC, we decided to use the existing resources to transfer reconfiguration information. This implies that virtual circuits dedicated for transmission of reconfiguration information must be set up alongside the virtual circuits that are used for transmission of regular data; for example, a reconfiguration channel from a master core to each of the other cores in the platform.

As mentioned earlier, reconfiguration of a NoC typically requires accessing and modifying some state in the NoC, as well as flushing the virtual circuits that are torn down and performing some initialization of virtual circuits that are set up. Our NoC is a source routed NoC with simple routers without any buffers, flow control or arbitration. For this reason, reconfiguration involves only the NIs.

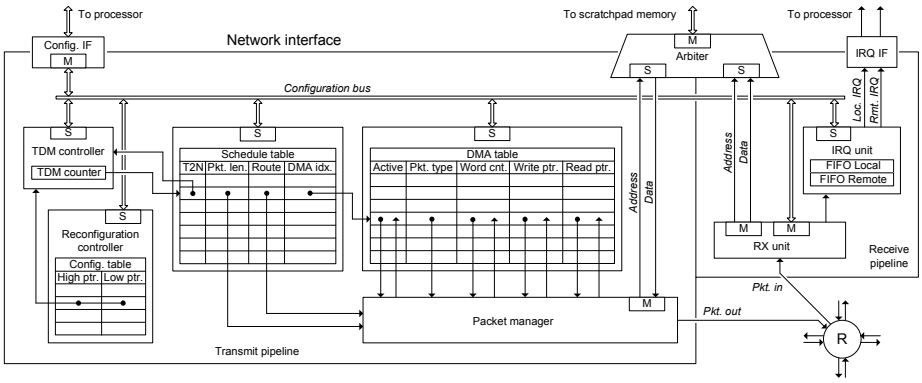


Figure 2.1: A block diagram of our NI. The block diagram is split in two parts: the transmit pipeline and the receive pipeline.

Moreover, the end-to-end transmission of packets for which incoming packets are written directly into the local SPM of the destination node, in combination with the way the scheduler maps virtual circuits to time slots in the TDM schedule, means that the network of routers is conceptually drained from packets at the end of each TDM period. This opens for the very interesting perspective of instantaneously switching from one TDM schedule to another in a way that is fully transparent to virtual circuits that persist across the reconfiguration. The last TDM period of one schedule can be followed immediately by the first TDM period of a new schedule, and the virtual circuits that persist across a mode change can be mapped to different paths and different timeslots in the TDM schedule, without any interference on the data flows. This again avoids the fragmentation of resources seen in the previously published solutions [5, 11] in which no changes can be made to virtual circuits that persist across a mode change.

The architecture of the NI is split into the transmit pipeline and the receive pipeline. The two pipelines interact through the SPM arbiter, the maximum rate of both pipelines is one 64-bit transaction every two cycles. Therefore, the arbiter can safely prioritize read requests from the transmit pipeline, with just a single register in the receive pipeline interface.

In the transmit pipeline, the TDM controller reads out scheduling information of the current time slot from the schedule table. The output from the schedule table activates the packet manager that assembles the packet header, updates the fields in the DMA table, reads the payload data from the local SPM, and sends the packet.

An important extension of our NI compared to Argo, is that incoming packets can be written to the internal tables of the NI, which allows a remote core to change the settings of the NI. We have extended the packet format to

3 types of packets, data packets, configuration packets, and interrupt packets. In the receive pipeline the RX unit handles the incoming packets according to their type.

2.4 Hardware Architecture

The packet format of the three packet types that are used in our NI, consists of a 32-bit header followed by *2n* 32-bit payloads. Fig. 2.1 shows our NI architecture. The transmit pipeline contains the TDM controller, the schedule table, the DMA table, the packet manager, and the reconfiguration controller. The reconfiguration controller is described in Section 2.5.

At boot time, each core initializes its schedule table and configuration table in the local NI. When all NIs are initialized, the master core signals all TDM counters to start counting synchronously, through a global reset signal.

The TDM controller indexes into the schedule table. The *T2N* (time-to-next) field contains the number of cycles until the TDM controller increments the index. The *Pkt. len.* field is the number of 64-bit payloads following the packet header. The *Route* field contains the route the scheduled packet must follow through the NoC. The *DMA idx.* field is the DMA table index of the associated DMA transfer.

The DMA table contains the DMA entries that the processor uses to transfer data through the NoC. The processor and the configuration unit can set the *Active* bit, to indicate that the DMA is active and the packet manager can clear the *Active* bit when the last data is sent. The transmit pipeline logic reads the active bit to determine whether a packet should be sent. The *Word cnt.* field is the number of 64-bit words remaining of the active DMA transfer. The *Read ptr.* points to the next local 64-bit word address of the DMA transfer. The *Write ptr.* is the write address for the next packet header. The *Pkt. type* is the packet type for the DMA transfer. When a packet is sent, *Word cnt.* is decremented and *Read ptr.* and *Write ptr.* are incremented all by the packet length.

The receive pipeline, shown in Fig. 2.1, contains the RX unit and the interrupt request (IRQ) FIFO. According to the packet type, the RX unit writes the packet payload to either the SPM, the internal tables of the NI, or the IRQ FIFO. The IRQ FIFO unit contains two queues that store the interrupt IDs of the interrupts. A local interrupt is invoked by the NI when the last data packet of a DMA transfer arrives at the destination to indicate the completion of a data block transfer. A remote in interrupt is invoked when an interrupt packet arrives at the NI.

2.5 Reconfiguration Controller

To support reconfiguration, we add a reconfiguration controller to the NI and connect the RX unit to the configuration bus, as shown in in Fig. 2.1. The latter allows the RX unit to write incoming configuration packets into all the tables connected to the configuration bus. The schedule table may hold multiple schedules at the same time, each spanning a range of entries. Each range is represented by a pair of pointers, high and low, stored in a small table in the reconfiguration controller. A reconfiguration simply requires that the TDM counter is set to the start entry of the new schedule when the TDM counter reaches the end of the current schedule.

A master processor invokes a reconfiguration of the NoC by sending a configuration packet to the reconfiguration controllers of all slave processor NIs, announcing that they must switch to the new schedule. This packet contains two parameters: the index of the reconfiguration table entry that holds the high and low pointers for the new schedule and the ID of the TDM period after which the new schedule should start.

The ID of the TDM period is defined in the reconfiguration controllers in all the NIs, but due to pipelining of routers, packets sent during a TDM period arrive in a time window that is phase-shifted by some clock cycles. Our scheduler minimizes this phase shift: virtual circuits that cross many routers are not scheduled towards the end of a TDM period. With this constraint, the phase shift is determined by the pipeline depth through the routers on the shortest path between two NIs. In our design, the shortest path is through two routers and, using 3-stage pipelined routers, this results in a phase-shift of 6 clock cycles. If a master processor issues a reconfiguration command in TDM period i, then the NI in the master node sends configuration packets to all the nodes during the next TDM period, $i + 1$. This means that some configuration packets may be received during TDM period $i+2$ and therefore the new schedule can start after TDM period $i + 2$ has finished.

2.6 Evaluation

This section evaluates the proposed architecture in terms of worst-case reconfiguration time (WCRT), hardware cost, and maximum operating frequency of the NI.

For the WCRT evaluation, we use a 4-by-4 platform with a bi-torus network and 3-stage pipelined routers. The WCRT of a new schedule C_{new} depends on the currently executing schedule C_{curr}. The worst-case analysis of software depends on the processor that executes the software. Therefore, we consider the software overhead of setting up DMA transfers outside the scope of this paper.

The preferred situation is that the C_{new} is already loaded in the schedule table of the slave NIs. It can be resident or pre-loaded in the background. In this case the WCRT is only three times the TDM period of C_{curr}, as explained in Section 2.5. For the benchmarks presented in Table 2.1, the WCRT is between 135 and 381 clock cycles, depending on the current benchmark.

In case that the schedule C_{new} is not already loaded in the schedule table of the slave NIs, we need to add the worst-case latency of transferring C_{new} to the slave NIs. For this we assume that C_{new} is loaded in the processor local SPM of the reconfiguration master. The worst-case latency of transferring C_{new} to the slave NIs is the maximum of the individual worst-case latency for each slave NI. We calculate the worst-case reconfiguration time between the schedules of the MCSL benchmark suite [7] and an All2all schedule to any other of these. The results are shown in Table 2.1. For space reasons, we leave out the column for H264-1080p, as the results are identical to H264-720p.

We see that the WCRT in Table 2.1 is between 549 and 2298 clock cycles. The Sparse benchmark, as C_{curr}, results in the lowest WCRT, as Sparse has the shortest TDM period, and thus the highest bandwidth to the slaves. The worst-case transfer time of C_{new} dominates the WCRT.

The time interval required by Æthereal and dAElite to set up a single virtual circuits is 246 and 60 clock cycles, respectively [11]. Moreover, these NoCs also need to tear down unnecessary virtual circuits. Assuming that any two modes differ by more than a handful of virtual circuits, the reconfiguration time of our NoC is in general comparable or shorter than the one of Æthereal and dAElite.

In case the new schedule needs to be transmitted to the slave NIs, our approach is still comparable. The maximum WCRT shown in Table 2.1 is 2298 clock cycles. For our NoC, this transition represents the transmission and the reconfiguration of 255 virtual circuits. In this time interval, Æthereal and dAElite can only set-up 9 and 38 virtual circuits, respectively.

We compare the hardware cost and the maximum operating frequency of the presented design against the TDM-based NoCs aelite and dAElite [11] as well as the original Argo NoC [6]. Table 2.2 shows the synthesis results of the four designs for one router and one NI and the supported number of TDM slots and connections per node. The published numbers we compare with are all a 2-by-2 mesh topologies implemented on the *Xilinx Virtex-6* FPGA architecture. We divided the numbers by four to get the hardware consumption of one 3-ported router and one NI.

The results in Table 2.2 for a 3-port router show that overall our implementation is smaller than the other NoCs against which we compare and has a similar maximum operating frequency f_{max}. In terms of slices, our implementation is a factor of 3.9 times smaller than aelite and a factor of 2.2 times smaller than dAElite. Table 2.2 also presents figures for a network node with a 5-ported router and a reasonable number of TDM slots and connections for

Table 2.1: The worst-case reconfiguration time, including schedule transfer. Since the worst-case reconfiguration time depends on the current schedule and the new one, we show a matrix of the combination of current and new schedules.

C_{curr} \ C_{new}	FFT-1024	Fpppp	RS-dec	RS-enc	H264-720p	Robot	Sparse	All2-all
FFT-1024	–	1436	1169	1080	1077	1074	991	1611
Fpppp	1627	–	1244	1149	1152	1149	677	1722
RS-dec	1593	1497	–	1125	1128	1125	1032	1680
RS-enc	1491	1401	1140	–	1056	1059	966	1572
H264-720p	1590	1494	1221	1128	–	1122	1029	1689
H264-1080p	1590	1494	1221	1128	1131	1122	1029	1689
Robot	2165	2041	1660	1539	1536	–	1406	**2298**
Sparse	777	684	594	**549**	552	**549**	–	822
All2all	1539	1452	1176	1086	1095	1092	1002	–

a larger platform. Comparing these results the Argo NoC we can see that our extended NoC is only around 17% larger than the original Argo NoC.

The results in Table 2.2 for the maximum operating frequency f_{max} show that the maximum frequency of our implementation is comparable to the ones of aelite and dAElite for a 3-port router. For a 5-port router, our implementation is around 30% faster than the 3-port router, since it uses block RAM instead of distributed memory (FFs), and around 10% faster than Argo.

2.7 Conclusion

This paper presented a resource-efficient NoC that supports reconfiguration for mode changes. The NoC extends the existing NI of the Argo NoC and provides guaranteed-service communication between processors. An implementation of the proposed architecture is evaluated in terms of worst-case reconfiguration time, hardware cost, and maximum operating frequency. The results show that our NoC is between 2.2 and 3.9 times smaller than NoCs with similar functionality and that the worst-case reconfiguration time is comparable or shorter to those NoCs.

Acknowledgment

The work presented in this paper was funded by the Danish Council for Independent Research | Technology and Production Sciences under the project RTEMP, contract no. 12-127600, (`http://rtemp.compute.dtu.dk`).

Table 2.2: A hardware resource and maximum operating frequency comparison of one router and one NI between the presented architecture and three similar designs.

	3-port router			5-port router	
	aelite	dAElite	Our imp	Argo	Our imp
Slots	8	8	8	256	256
Conn	1	1	2	64	64
Slices	1375	774	350	289	338
LUTs	1916	2506	1279	1119	1267
FFs	3861	3081	1074	871	924
BRAM	0	0	0	4	4
f_{max} (MHz)	119	122	124	146	161

Bibliography

[1] L. Benini, E. Flamand, D. Fuin, and D. Melpignano. P2012: Building an ecosystem for a scalable, modular and high-efficiency embedded computing accelerator. In *Proc. Design, Automation and Test in Europe (DATE)*, pages 983–987, 2012.

[2] D. Berozzi. Network interface architecture and design issues. In G. DeMicheli and L. Benini, editors, *Networks on Chips*, chapter 6, pages 203–284. Morgan Kaufmann Publishers, 2006.

[3] A. Burns and A. Wellings. *Real-Time Systems and Programming Languages: Ada 95, Real-Time Java and Real-Time POSIX*. Addison-Wesley, 2001. ISBN 0201729881, 9780201729887.

[4] F. Clermidy, C. Bernard, R. Lemaire, J. Martin, I. Miro-Panades, Y. Thonnart, P. Vivet, and N. Wehn. A 477mW NoC-based digital baseband for MIMO 4G SDR. In *Proc. IEEE Intl. Solid-State Circuits Conference Digest of Technical Papers (ISSCC)*, pages 278–279, 2010. doi: 10.1109/ISSCC.2010.5433920.

[5] K. Goossens and A. Hansson. The aethereal network on chip after ten years: Goals, evolution, lessons, and future. In *Proc. ACM/IEEE Design Automation Conference (DAC)*, pages 306 –311, June 2010.

[6] E. Kasapaki, M. Schoeberl, R. B. Sørensen, C. T. Müller, K. Goossens, and J. Sparsø. Argo: A Real-Time Network-on-Chip Architecture with an Efficient GALS Implementation. *IEEE Transactions on VLSI Systems*, 24 (2):479–492, 2015.

[7] W. Liu, J. Xu, X. Wu, Y. Ye, X. Wang, W. Zhang, M. Nikdast, and Z. Wang. A NoC traffic suite based on real applications. In *VLSI (ISVLSI), 2011 IEEE Computer Society Annual Symposium on*, pages 66 –71, july 2011.

[8] A. Radulescu, J. Dielissen, S. Pestana, O. Gangwal, E. Rijpkema, P. Wielage, and K. Goossens. An efficient on-chip NI offering guaranteed services, shared-memory abstraction, and flexible network configuration. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 24(1):4–17, 2005. ISSN 02780070. doi: 10.1109/TCAD.2004.839493.

[9] R. B. Sørensen, J. Sparsø, M. R. Pedersen, and J. Højgaard. A metaheuristic scheduler for time division multiplexed networks-on-chip. In *Proc. IEEE/IFIP Workshop on Software Technologies for Future Embedded and Ubiquitous Systems (SEUS)*, pages 309–316, 2014.

[10] J. Sparsø, E. Kasapaki, and M. Schoeberl. An area-efficient network interface for a TDM-based network-on-chip. In *Proc. Design, Automation and Test in Europe (DATE)*, pages 1044–1047, 2013.

[11] R. A. Stefan, A. Molnos, and K. Goossens. dAElite: A TDM NoC Supporting QoS, Multicast, and Fast Connection Set-Up. *IEEE Transactions on Computers*, 63(3):583–594, 2014. ISSN 00189340, 15579956. doi: 10.1109/TC.2012.117.

Chapter 3

A Resource-Efficient Network Interface Supporting Low Latency Reconfiguration of Virtual Circuits in Time-Division Multiplexing Networks-on-Chip

---

This chapter is submitted to Elsevir Journal of Systems Architecture as:
Rasmus Bo Sørensen, Luca Pezzarossa, Martin Schoeberl, and Jens Sparsø,
"A Resource-Efficient Network Interface Supporting Low Latency Reconfiguration of Virtual Circuits in Time-Division Multiplexing Networks-on-Chip".

Abstract

This paper presents a resource-efficient time-division multiplexing (TDM) network interface (NI) of a network-on-chip (NoC) intended for use in a multicore platform for hard real-time systems. The NoC provides virtual circuits (VCs) to move data between core-local on-chip memories. In such a platform, a change of the application's operating mode may require reconfiguration of VCs that are setup by the NoC. A unique feature of our NI is the instantaneous reconfiguration between different TDM schedules, containing sets of VCs, without affecting VCs that persist across the reconfiguration. The results show that the worst-case latency from triggering a reconfiguration until the new schedule is executing, is in the range of 300 clock cycles. Experiments show that new schedules can be transmitted from a single master to all slave nodes for a 16 core platform in between 500 and 3500 clock cycles. The results also show that the hardware cost for an FPGA implementation of our architecture is considerably smaller than other NoCs with similar reconfiguration functionalities, and that the worst-case time for a reconfiguration is smaller than that seen in functionally equivalent architectures.

3.1 Introduction

Packet-switched networks-on-chip (NoCs) have become the preferred paradigm for interconnecting the many cores (processors, hardware accelerators etc.) found in today's complex application-specific multi-processor systems-on-chip [7, 1] and general-purpose chip multi-processors [6, 11].

In the multi-processor systems-on-chip domain, a significant amount of previous research has targeted the generation of application-specific NoC platforms e.g., [21, 3]. With the growing cost of developing and fabricating complex VLSI chips, application-specific platforms are only feasible for very few ultra-high-volume products. In all other cases, a cost-efficient platform must support a range of applications with related functionality. This implies that the hardware resources and the functionality they implement should be as general-purpose and generic as possible, targeting a complete application domain instead of a single application. This view is expressed in the principle *provide primitives not solutions* that is well-known and accepted in the field of computer architecture. We adopt this view, striving to avoid hardware resources for dedicated and specific functionality.

The application domain we target is real-time systems. In real-time systems, the whole architecture needs to be time-predictable to support worst-case execution time (WCET) analysis. A NoC for real-time systems needs to support guaranteed-service (GS) channels. Furthermore, many hard real-time applications have multiple modes of operation. To support applications that change between operating modes, the NoC must be able to reconfigure the virtual circuits (VCs) at run-time.

This paper proposes and evaluates a flexible and resource-efficient network interface (NI) for hard real-time systems. Our NoC implements VCs using static scheduling and time-division multiplexing (TDM). A VC provides GS channels in the form of a guaranteed minimum bandwidth and a maximum latency. Furthermore, transfer of data between an on-chip memory and the NoC is coupled with the TDM schedule so that we can give end-to-end guarantees for the movement of data from one core-local memory to another core-local memory. This architecture avoids both *physical* VC buffers in the NIs and credit-based flow control among the NIs that are found in most other NoC designs [2, 23, 28]. Moreover, the usage of TDM schedules leads to a reduced hardware complexity due to the lack of buffering in the routers and due to a static traffic arbitration.

The main contribution of the paper and a key feature of this NI is its very efficient support for mode changes. The active schedule can be switched from one TDM period to the next, without breaking the communication flow of VCs that persist across the switch. This is in contrast to the Æthereal family of NoCs [12, 28], which provides similar functionality at a higher hardware cost and longer reconfiguration time.

Our NI can store multiple TDM schedules and it supports instant switching from one schedule to another, synchronously across all NIs. The last TDM period of one schedule can be followed immediately by the first TDM period of a new schedule. This allows VCs that persist across a schedule switch to be mapped to different paths, without any interference to their data flow. This again avoids the fragmentation of resources seen in the previously published solutions [12, 28], in which no changes can be made to circuits that persist across a mode change and where the set-up of a new circuit is limited to using free resources.

If the schedule tables are too small to store all necessary schedules, our NoC can transparently transmit new schedules via the standard VCs. In this way we avoid fixed allocation of resources for schedule transmission.

The NI presented here is an extension of [27], which is part of the Argo NoC [16]. A preliminary version of the new NI was published in [26]. In the rest of the paper, we refer to the NoC that uses the new NI as the Argo 2.0 NoC. The main contributions of this paper are:

- support of instant reconfiguration of VCs;
- a more elaborate analysis of the TDM schedule distribution through the NoC;
- variable-length packets to reduce the packet header overhead, resulting in shorter schedules and/or higher bandwidth on the VCs;
- interrupt packets to support multicore operating systems;
- a more compact TDM schedule representation in the NIs, reducing the schedule memory requirements;
- analysis of the effect on the TDM period length of using GS communication for reconfiguration;
- a discussion on the scalability of the architecture.

This paper is organized into seven sections. Section 3.2 presents related work. Section 3.3 provides background on mode changes, TDM scheduling, and the Argo NoC. Section 3.4 presents the Argo 2.0 architecture in detail. Section 3.5 describes the reconfiguration method and its utilization. Section 3.6 evaluates the presented architecture. Section 3.7 concludes the paper.

3.2 Related Work

This section presents a selection of NoCs that offer GS connections and that support run-time reconfiguration of the GS provided. One approach to implementing GS connections is to use non-blocking routers in combination with

mechanisms that constrain packet injection rates. These NoCs are reconfigured by resetting the parameters that regulate the packet injection rates to the new requirements.

Mango [4] uses non-blocking routers and rate-control, but only links are shared between VCs. Each end-to-end connection is allocated to a unique buffer in the output port of every router that the connection traverses and these buffers use credit-based flow control between them. The bandwidth and latency of the different connections are configured by setting priorities in the output port arbiters of the router and by bounding the injection rate at the source NI. Connections are set up and torn down by programming the crossbar switches, which is done using best effort (BE) traffic. In Mango, we can observe that the reconfiguration directly interacts with the rate control mechanism in the NIs, the crossbars, and the arbiters in the routers. In addition, the fact that GS connections are programmed using BE packets may compromise the time-predictability of performing a reconfiguration.

The NoC used in the Kalray MPPA-256 processor [10] uses flow regulation, output-buffered routers with round-robin arbitration, and no flow control. Network calculus [8] is used to determine the flow regulation parameters that constrain the packet injection rates such that buffer overflows are avoided and GS requirements are fulfilled. The Kalray NoC is configured by initializing the routing tables and injection rate limits in the NIs.

IDAMC [20] is a source-routed NoC using credit-based flow control and virtual channel input buffers together to provide GS. IDAMC provides GS connections by implementing the Back Suction scheme [9], which prioritizes non-critical traffic while the critical traffic progresses to meet the deadline.

To our knowledge, details on how reconfiguration is handled in Kalray, Mango and IDAMC have not been published. However, we can safely assume that setting up a new connection must involve the initialization and modification of flow regulation parameters, and tearing down a connection must involve draining in-flight packets from the VC buffers in the NoC.

An alternative to the usage of non-blocking routers in combination with constrained packet injection rates is VC switching implemented using static scheduling and time-division multiplexing (TDM). These NoCs can be reconfigured by modifying the schedule and routing tables in the NIs and/or in the routers.

The Æthereal family of NoCs [12, 28] uses TDM and static scheduling to provide GS. The original Æthereal NoC [13] supports both GS and BE traffic. The scheduling tables are in the NIs and the routing tables are in the routers. Reconfiguration is performed by writing into these tables using BE traffic. Analogously to the Mango NoC approach, this way of doing things may compromise the time-predictability of a (re)configuration. The dAElite NoC [28] focuses on multicast and overcomes this problem by introducing a separate dedicated NoC

with a tree topology for the distribution of the schedule and routing information during run-time reconfiguration.

The aelite NoC [12] only supports GS traffic and it is based on source routing. This reduces significantly the high hardware cost of distributed routing and combined support for BE and GS traffic of the original Æthereal NoC. For this NoC, the routers are simple pipelined switches and both schedule tables and routing tables are in the NIs. Reconfiguration involves sending messages across the NoC using GS connections from a reconfiguration master to the schedule and routing tables that are required to change; these GS connections are reserved for this purpose only.

The original version of the Argo NoC [16] has some functional similarity with aelite. It only supports GS traffic and it also uses a TDM router with source routing. The Argo design avoids VC buffers and the credit-based flow control that account for most of the area of the NIs of the Æthereal, aelite, and dAElite range of NoCs. The Argo NoC uses a more efficient NI [27] in which the direct memory access (DMA) controllers are integrated with the TDM scheduling. The original version of the Argo NoC does not support reconfiguration.

In all the presented NoCs that uses VC switching and TDM static scheduling, the re-mapping of VCs that persist across the reconfiguration is not supported, since the reconfiguration is done incrementally (tearing down unused circuit and setting up new ones). This can lead to sub-optimal usage of resources due to fragmentation. If re-mapping of VCs is needed, the entire application must be suspended during the reconfiguration.

This paper presents a new version of the Argo architecture that implements the same functionality as the first version of Argo, while adding instantaneous reconfiguration capabilities, including re-mapping of VCs that persist across the reconfiguration.

3.3 Background

This section provides background on the Argo NoC, TDM scheduling, and reconfiguration for mode changes.

3.3.1 Message passing in the Argo NoC

Argo is a packet-switched and source-routed NoC that uses static allocation of network resources through TDM as a means to provide VCs for which communication bandwidth and latency can be guaranteed.

The NoC offers message passing communication. Technically, this is implemented using DMA controllers, one per source end of every VC. A DMA controller transfers a block of data from the local memory of a processor node into the local memory of a remote node. This functionality is similar to what

is seen in many other multi-core platforms including the Cell processor [17], the CompSoC platform [14], and the Epiphany processor [22]. Argo uses a very efficient NI-architecture [27] in which the DMA controllers have been integrated with the TDM-mechanism in the NI. This integration avoids all the buffering and flow control that is found in most NoCs. In addition, the NI-hardware is dominated by area-efficient memory structures in the form of configuration tables.

3.3.2 TDM Scheduling

A parallel application on a multicore platform can be described as a set of tasks mapped to a set of processors. The steps of mapping a real-time application onto a multi-core platform and the generation of a TDM schedule for the T-CREST platform are shown in Fig. 3.1.

A set of communicating tasks can be modeled as a *task graph* (Fig. 3.1(a)), where the vertices represent the tasks and the edges represent the communication between them.

By assigning the tasks to the processing nodes, it is possible to derive a *core communication graph* (Fig. 3.1(b)). The assignment of tasks to processing nodes has to be performed in a way that minimizes the total number of hops for traffic. For this graph, the vertices represent the processing nodes, and the edges represent the set of VCs between each pair of processing nodes.

TDM scheduling shares the resources of the NoC in time between multiple VCs. The Argo NoC uses the scheduler described in [25]. This approach divides the time into TDM periods, and a period is further divided into time-slots.

The scheduler is an off-line procedure that uses the bandwidth requirements and a description of the NoC topology to generate a schedule that avoids deadlocks and collisions, and that ensures in-order arrival of packets. The static schedule is stored in the NIs of the NoC and specifies the route of each packet and the time slot in which each packet is injected into the router. We can calculate the minimum frequency that the schedule should run at, from the bandwidth requirements and the created schedule.

Fig. 3.1(c) shows (part of) two TDM periods for the traffic out of the processor P_0 (VC *c1* and *c4*). VC *c1* has been assigned four time slots and VC *c4* has been assigned two times three time slots which will use two different (shortest) paths through the NoC (*c4'* and *c4"*). Fig. 3.1(d) shows the VC paths on a section of the multi-core platform.

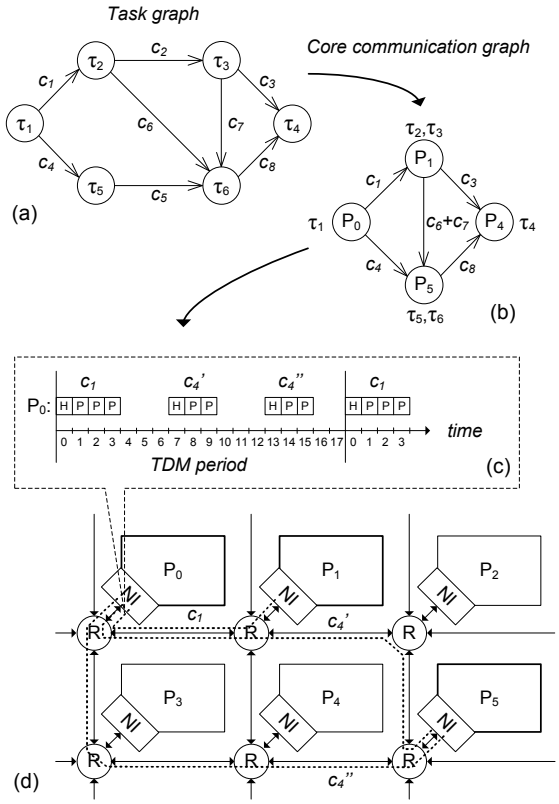


Figure 3.1: Mapping of an application onto a multi-core platform: (a) task graph for an application, (b) core communication graph, (c) TDM schedule for processor P_0, and (d) section of multi-core platform with possible routing for processor P_0.

3.3.3 Argo NoC Architecture

As already mentioned, in the Argo NI architecture, the TDM-driven DMA controllers are integrated into the NI. This avoids buffering and flow control and leads to an efficient NI architecture.

Fig. 3.2 shows a 2-by-2 section of a regular mesh topology, and the expanded tile in the figure shows the interface between the processor and the NoC as well as key elements of the NI. The processor is connected to one side of a dual-ported scratchpad memory (SPM), and the NI is connected to the other side of the SPM. The SPM populates a part of the processor's local address space and the processor sees it as a regular data SPM.

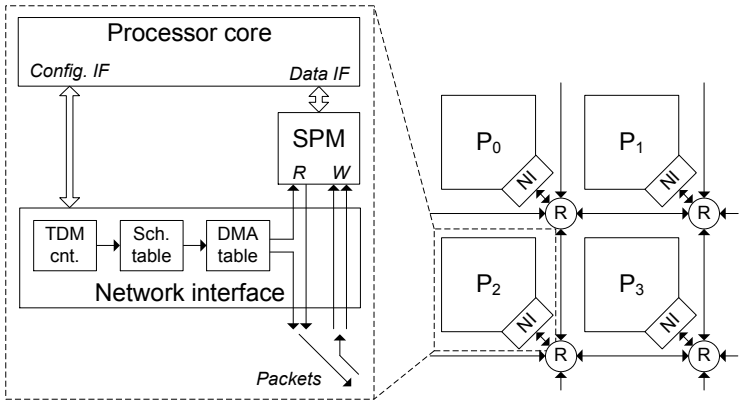


Figure 3.2: A 2-by-2 section of a multicore platform and the content of a processing tile.

Each NI contains a *TDM counter*, that indexes into a *schedule table*, see Fig. 3.2. The value of the TDM counter is not the current TDM slot, but it is the index of the current schedule table entry. The TDM counters in all the NIs operate synchronously and wrap around at the end of the TDM period. Each entry in the schedule table points to an entry in the *DMA table* that stores the counters and pointers corresponding to a DMA controller, and the route that a packet should follow through the network. The indexed DMA controller reads the payload data of a packet from the SPM, illustrated in Fig. 3.2, and sends a packet. The fact that the DMA controller is activated by the TDM counter means that the DMA controller reads the data from the SPM just in time to transmit it across the network. Finally, when a packet is received at its destination NI, the payload is directly written into the SPM at the target address carried by the packet.

The Argo router is a pipelined crossbar that routes incoming packets according to the routing information contained in the packet header. Argo supports both synchronous, mesochronous, and asynchronous router implementations. For the Argo 2.0 NoC, we assume a 3-stage synchronous implementation of the router as shown in Fig. 3.3. However, the NoC is compatible with any of the Argo routers [15, 18]. The router shown in Fig. 3.3, consist of the three pipeline stages: link traversal, header passing unit (HPU), and crossbar. The header of an incoming packet is read in the HPU and, based on the route in the header, the packet is routed to the output port in the crossbar stage.

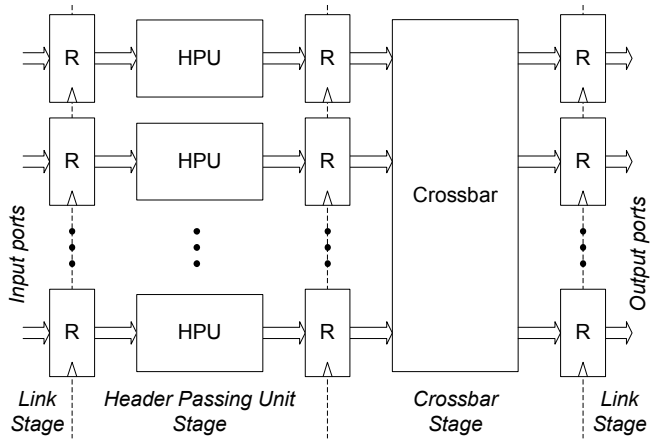


Figure 3.3: An Argo router with three pipeline stages and registers (R). The three pipeline stages are the link stage, the header passing unit (HPU) stage, and the crossbar stage.

3.3.4 Reconfiguration for Mode Changes

Finally, we provide some background on reconfiguration for mode changes, since this is the main contribution of this paper.

In a parallel application, a mode change is defined as a change in the subset of the executing software tasks during normal operation. Mode changes can be triggered as part of the normal operation of the system or in response to external events [5, p.340]. In normal operation, a mode change is triggered at a well-defined moment in the application execution. As a response to an external event, a mode change is triggered to adapt the system behavior to new environmental conditions. For example, an external alarm may require the execution of a set of tasks to manage specific situations.

Real-time multicore applications rely on the GS communication of the NoC to guarantee correct timing behavior. A single configuration of the time-predictable NoC may be unable to support all modes of operation for a given real-time application. Such applications need the time-predictable NoC to support reconfiguration of the VCs during run-time. This reconfiguration needs to be performed in bounded time to guarantee correct behavior of the tasks that continue operating across a mode change of the application.

We assume that each mode consists of a set of communicating tasks assigned to processors and that each mode has an associated core communication graph, from which the TDM scheduler can generate a schedule for the corresponding configuration.

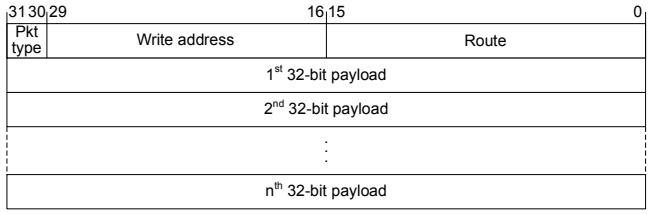


Figure 3.4: Argo 2.0 network packet format. A packet contains one 32-bit header word and n 32-bit payload words. For configuration and interrupt packets, $n = 1$ and for data packets $n = 1, 2, \ldots, 15$.

3.4 Argo 2.0 Microarchitecture

Before discussing reconfiguration, we first present the basic operation and microarchitecture of the Argo 2.0 NI. Compared to the original Argo NI [27] the Argo 2.0 NI has a more elaborate microarchitecture that allows a more compact representation of a TDM schedule. The following four subsections describe the Argo 2.0 packet format, the compact schedule representation, and how the Argo 2.0 NI design transmits and receives packets.

3.4.1 Packet Format

The microarchitecture of the Argo 2.0 NI supports three types of network packets: data packets, interrupt packets, and configuration packets. Fig. 3.4 shows the general packet format, it contains a 32-bit header followed by n 32-bit payloads. For configuration and interrupt packets, $n = 1$ and for data packets $n = 1, 2, \ldots, 7$. The variable length of data packets that allow quite long packets may be used to reduce the header overhead for VCs that require high bandwidth. The most significant two bits of the header contain the packet type. The next 14 header bits contain the write address in the target SPM where the payload data of the packet will be written. The last 16 header bits contain the route that the packet will take through the NoC.

Data packets are used to transfer regular data from the local SPM of one core to the local SPM of another core. A single DMA transfer may involve a sequence of packets sent during several consecutive TDM periods. If the sender process needs to notify the receiver when the DMA transfer is complete, the sender can mark the transfer so the last packet invokes an interrupt at the destination core. We call this a local interrupt, as it is generated and processed in the processor node that receives the message.

Interrupt packets are used to invoke an interrupt in a remote processor core, and this feature is needed to support multicore operating systems. When

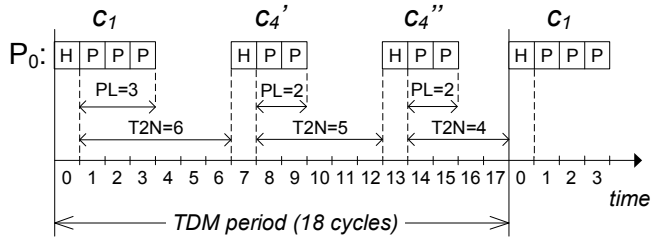


Figure 3.5: Detailed view of how the schedule in Fig. 3.1(c)is represented in the NI. 'PL' stands for packet length.

an interrupt packet arrives at the remote core it invokes an interrupt. We call this a remote interrupt, as it is triggered by a remote core.

Configuration packets are used to remotely write configuration data into the tables of a remote NI. The data of a configuration packet is written word by word into the tables of the NI.

3.4.2 Compact Schedule Representation

The Æthereal family of NoCs and the original Argo NoC use a fixed 3-word packet format. In both designs the TDM counter is incremented once every 3 clock cycles, resulting in a 3 clock cycle slot. The TDM counter in these designs index directly into the schedule table, where unused entries are marked as *not valid*. Compared to these relative straightforward designs, the Argo 2.0 NI design represents the schedule in a more compact form. Argo 2.0 represent each packet with an entry in the schedule table and adds two fields to each entry of the schedule table. One of these fields specifies the number of payload words of the specific packet and the other specifies the time until the header of the next packet; an example of this is illustrated in Fig. 3.5. In the example, the schedule period is 18 clock cycles and the schedule requires 3 entries in the schedule table. For comparison, we mention that the original Argo and the Æthereal family would require 6 entries in the schedule table in order to represent a schedule with a period of 18 cycles (6 TDM slots of 3 cycles each).

The incremental reconfiguration that is used by the Æthereal family requires this uncompressed representation, such that a scheduled packet can be written into the active schedule in one atomic write. Because the Argo 2.0 reconfiguration approach can instantaneously switch between two configurations, we can compress the schedule.

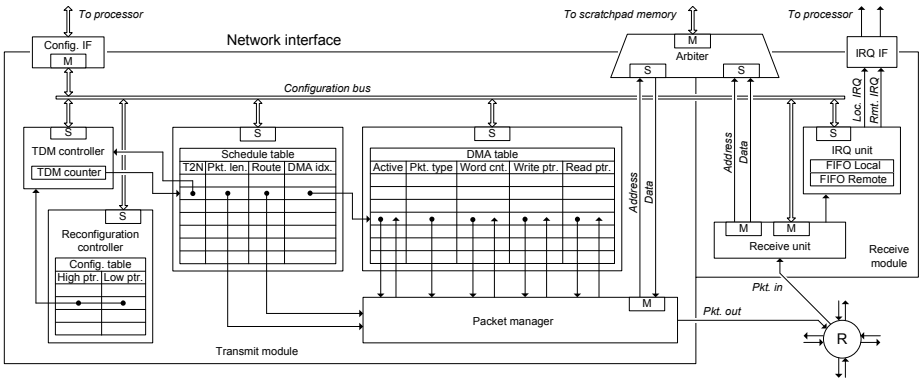


Figure 3.6: A block diagram of our NI. The block diagram is split into two parts: the transmit module and the receive module.

3.4.3 Transmitting Packets

The transmit module of the NI, shown in Fig. 3.6, consists of the following components: the TDM controller, the schedule table, the DMA table, the packet manager, and the reconfiguration controller. The reconfiguration controller is described in Section 3.5.

The NI contains several tables where pointers in one table index into the next. Its architecture and operation are best explained using an example. Fig. 3.1(c) showed a schedule executed by the NI in processor node P_0 that has two outgoing VCs *c1* and *c4*. Fig. 3.5 shows the same schedule in greater detail. The scheduler has assigned one 4-word packet per TDM period to *c1* and two 3-word packets to *c4*. The period of the TDM schedule in the example is 18 clock cycles, and in principle the TDM counter operates as a wrapping modulo-18 counter.

The example schedule requires three entries in the schedule table, one for *c1* and two for *c4*. Passing time in combination with the time-to-next (T2N) field controls how and when the TDM controller indexes/accesses the schedule table. In the example, the TDM controller will access the entry corresponding to *c1* in TDM cycle 0, the entry corresponding to *c4'* in cycle 7 and the entry corresponding to *c4"* in cycle 13. As the implementation is pipelined, the table is actually accessed a few cycles earlier.

The schedule table can hold entries belonging to different schedules used for reconfiguration. The reconfiguration controller marks the region in the schedule table of the currently active schedule.

An entry in the schedule table contains the route of the packet that the NI is about to send. The entry also contains an index into the DMA table. Each entry in the DMA table represents a VC. Our example schedule with two

VCs requires two entries in the DMA table, one for the DMA controller that pushes data across *c1* and one for the DMA controller that pushes data across *c4*. Using information from the schedule table and from the DMA table the packet manager assembles and sends out packets. The header of an outgoing packet is assembled from the packet type field of the DMA table entry, the route field of the schedule table entry and the write address field of the DMA table entry. The following payload words are read from the SPM. The packet length field *Pkt. len.* of the schedule table entry indicates the maximum number of payload words that can be appended the header word. The words are read using the read address field of the DMA table entry. During transmission of a packet the read address, the write address and the word count in the DMA table are updated.

If the data DMA transfer is marked as causing a local interrupt when it completes, the NI marks the last packet when the word count field in the DMA table entry reaches zero. When the DMA transfer is complete the packet manager set the active field in the DMA entry to inactive. A TDM schedule reserves slots for the different VCs and the schedule table repeatedly indexes the DMA table accordingly. If the indexed DMA table entry is inactive, no packet is transmitted in the reserved slot.

As our scheduler [25] generates schedules with shortest-path routing, all the possible paths that a packet can take through the NoC have the same number of hops. This means that packets arrive in order, and this gives the scheduler the freedom to route multiple packets belonging to the same VC along different paths. For this reason, the Argo 2.0 design places the route field into the schedule table. In the example illustrated in Fig. 3.1 and Fig. 3.5 this feature may be used for VC *c4* where packets *c4'* and *c4"* may be sent along different routes.

In order to program a TDM schedule into the NIs, information must be written into the TDM controller, the schedule table and the reconfiguration controller of every NI. This can be done by the local processors or by a remote master processor sending out configuration packets as explained in the next subsection. The entries in the DMA table can be written and read by the local processor.

3.4.4 Receiving Packets

The receive module shown in Fig. 3.6 consists of two blocks: the receive unit and the interrupt (IRQ) unit. The receive unit processes incoming packets depending on the packet type. Incoming data packets carry the target address as part of the header and the data payload is written directly into the SPM as it is being received. For each packet, the receive unit increments the target address for each write into the SPM. If the data packet is the last packet of a DMA transfer, the target address of the last word is written into the IRQ FIFO for local interrupts.

If the received packet is a configuration packet, the data payload is written into one of the NI tables in the transmit module. The data structures in these blocks are mapped into a private address space of the NI and the address of the configuration packet header points into this address space.

If the received packet is an interrupt packet, the data payload is written into the SPM and the target address is written into the remote interrupt FIFO.

The IRQ unit contains two FIFO queues that store interrupts. One queue is for external interrupts communicated using the interrupt packet format. The other is for local interrupts that are generated when the last packet belonging to a message is received.

The transmit and receive modules share one port to the SPM. To allow sustained and concurrent 32-bit reads and writes, the SPM uses a double width read/write port. The associated buffering and arbitration is implemented by the SPM arbiter.

The data payload of incoming packets is written directly to its target address. Therefore, there is no need for buffers or flow control in the NI, or for extra DMA controllers in the processor to copy the received data out of the NI. This makes the area of the receive module very small.

3.5 Reconfiguration

This section describes how we support mode changes by reconfiguration. Firstly, we present the underlying observations and ideas, and introduce the architectural features supporting reconfiguration. Secondly, we discuss a number of ways in which the reconfiguration mechanism can be used by an application requiring mode changes.

3.5.1 Key Observations and Ideas

As we target domain-specific platforms that support a multitude of applications, our primary concern is to avoid adding resources that are specialized for one use. Therefore, we decided to use the available NoC for reconfiguration commands and transmission of schedules. This is in contrast to a dedicated (re)configuration network, as for example used in dAElite [28]. Given a fixed amount of hardware resources for the NoC, a dedicated reconfiguration NoC establishes a static split of bandwidth between regular traffic and configuration traffic. We prefer to use all hardware resources to provide as much total bandwidth as possible, leaving it to the application programmer to allocate bandwidth for schedule transmission and regular traffic.

In Argo 2.0, this implies that the VCs dedicated for reconfiguration commands and possible transmission of new schedules must be set up alongside the VCs that are used for transmission of regular data; for example a VC for re-

configuration from a master core to each of the other cores in the platform. As seen in the results section, the addition of VCs for configuration often has little impact on the TDM schedule period of an application.

As mentioned earlier, reconfiguration of a NoC typically requires accessing and modifying the state in the NoC, as well as flushing the VCs that are torn down and some initialization of VCs that are set up. In this respect, the Argo 2.0 NI has three characteristics that both individually and in combination greatly simplify reconfiguration.

Firstly, the combination of TDM and source routing means that the routers are simple, pipelined switches, without any buffers, flow control, or arbitration. A router does not preserve any state when switching between VCs. For this reason, reconfiguration does not involve the routers; only the NIs.

Secondly, Argo 2.0 avoids VC buffers in the NIs and credit-based flow control among these buffers. Therefore, Argo 2.0 does not need to flush VCs and initialize credits counters when new connections are set up.

Thirdly, end-to-end transmission of packets, in which incoming packets are written directly into the destination SPM, in combination with the way the scheduler maps VCs to time slots in the TDM schedule, means that the network is conceptually empty at the end of each TDM period. This opens for the very interesting perspective of instantaneously switching from one TDM schedule to another in a way that is fully transparent to VCs that persist across the reconfiguration. These circuits can even be re-mapped to different TDM slots and different (shortest path) routes. This feature avoids the fragmentation of resources that is seen in NoCs where VCs are torn down and created on an incremental basis. This ability to switch from one TDM schedule to another can be used to support reconfiguration and mode changes in a number of ways, as described at the end of this section.

3.5.2 Reconfiguration Controller

To support reconfiguration, we add a reconfiguration controller to the NI and connect the receive unit to the configuration bus, as seen in Fig. 3.6. Connecting the receive unit to the configuration bus allows the receive unit to write incoming configuration packets into all the tables connected to the configuration bus. The schedule table may hold several different schedules, each spanning a range of entries. Each range is represented by a pair of pointers, high and low, that are stored in the configuration table of the reconfiguration controller. A reconfiguration simply requires that the TDM counter is set to the start entry of the new schedule when the TDM counter reaches the end of the current schedule.

A master invokes a reconfiguration of the NoC by sending a reconfiguration packet to the reconfiguration controller of all the slave NIs, announcing that they must switch to the new schedule. This packet contains two parameters: the index of the reconfiguration table entry that holds the high and low pointers

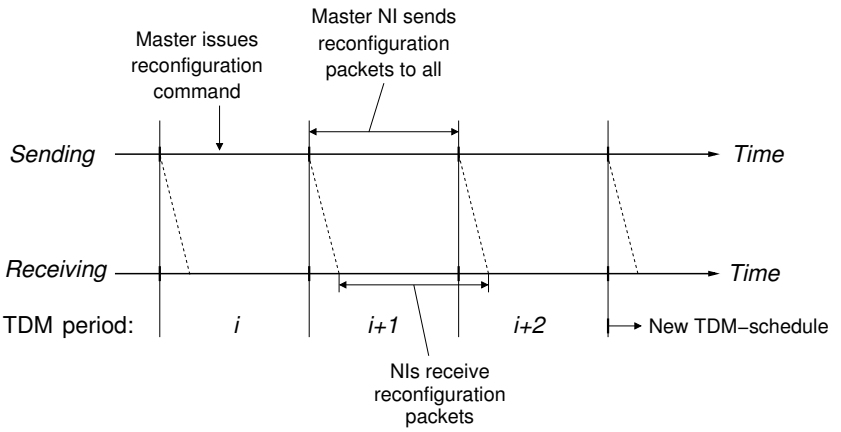


Figure 3.7: Time diagram illustrating the reconfiguration of the NoC.

for the new schedule and the number of the TDM period after which the new schedule should be used.

The reconfiguration process is illustrated in Fig. 3.7. When the master issues the reconfiguration command in TDM period i, the NI transmit configuration packets to all the slave reconfiguration controllers in period $i + 1$. Due to pipelining in the routers and in the NI, packets sent during a TDM period arrive in a time window that is phase-shifted by some clock cycles. Therefore, some packets will reach the final destination in TDM period $i + 2$. A new TDM schedule can thus be used at the earliest in TDM period $i + 3$.

All the tables that contain configuration data in the NI are connected to the receive unit through the configuration bus. The receive unit writes the incoming NoC configuration packets into these tables. Therefore, we can also use the NoC to transmit new schedules from the master core to the slave cores by sending the schedules using configuration packets. This transmission is transparent to the slave core.

3.5.3 Using the Reconfiguration Features

The reconfiguration mechanism described above can be used to implement reconfiguration in a number of ways when an application requests a reconfiguration:

1. In cases where the schedule table and the DMA table have sufficient capacity to store all possible configurations, these can be loaded into the NIs when the platform is booted. In this way, a master only needs to send reconfiguration requests to the NIs, and this method has the lowest reconfiguration latency.

2. Another approach is first to transmit the new schedule and then send a reconfiguration request. As seen in the next section, the time required to first distribute and then activate a new schedule is relatively short and comparable to the reconfiguration seen in other NoCs.

3. A hybrid of the above two methods is also possible, and is preferred if 1) is not possible. This hybrid divides the mode change graph into components, such that each component can be mapped into a statically sized portion of the schedule table. The mode change graph is divided by cutting either the least likely mode transitions or the mode transitions with the longest timing requirements. In this way, the reconfiguration master can switch rapidly between the schedules of one group. Switching between groups of schedules will include the transmission of the new group.

4. It is also possible to use the incremental approach [28], by tearing down and setting up individual circuits by writing into the live schedule. This approach requires a non-compacted schedule, where T2N and Pkt. len. are 3 and empty slots are represented by an invalid schedule table entry. As mentioned earlier, this method can suffer from fragmentation in the schedule tables.

Not all of these procedures are feasible for all applications, but the best solution is to use as much of the schedule table as possible. In general, this reduces the worst-case reconfiguration time.

3.6 Evaluation

This section evaluates the proposed architecture in terms of six criteria: (i) the TDM period extension due to statically allocating VCs for reconfiguration, (ii) the impact of variable-length packets on the schedule period, (iii) the storage size of the schedule in the schedule table, (iv) the worst-case reconfiguration time, (v) the worst-case schedule transmission time, (vi) and the hardware cost and maximum operating frequency of the NI. Each criterion is evaluated in one of the following subsections. For the evaluation, we use a 4-by-4 platform with a bi-torus network and 3-stage pipelined routers. We use the MCSL benchmark suite [19] and an All2all schedule as communication patterns for the evaluation. For space reasons, we leave out the H264-1080p benchmark, as its communication pattern is identical to that of H264-720p.

3.6.1 Virtual Circuits For Configuration

This subsection evaluates the TDM period extension due to statically allocating VCs for configuration packets in the application-specific schedules from the MCSL benchmark and an All2all schedule. In the All2all schedule each core

Benchmark	Baseline	w/ VCs for config.	
	(cc)	(cc)	(%)
FFT-1024	63	78	24
Fpppp	120	120	0
RS-dec	90	92	2
RS-enc	84	86	2
H264-720p	90	92	2
Robot	171	171	0
Sparse	27	30	11
All2all	54	75	39

Table 3.1: TDM period in clock cycles (cc) and overhead (%) relative to the baseline, when VCs are used for configuration.

communicates with equal bandwidth to all other cores. Table 3.1 shows the TDM period of the baseline schedules without VCs for configuration and of the baseline schedules plus the added VCs for configuration. The VCs for configuration connect the master core to all slave cores in the platform.

The traffic patterns from the MCSL benchmarks are mapped to homogeneous platforms of square dimension. We choose the core with the smallest outgoing bandwidth as the reconfiguration master core. We believe that this mimics a real application best, since in most cases the reconfiguration master would not have a high communication load.

A TDM schedule involving a master core that sends a single configuration packet to each slave core in the platform has a lower input/output (I/O) bound, on the TDM period, of two words per slave core. This is because the master needs to transmit for two clock cycles per slave core. For a 16 core platform this lower I/O bound on the TDM period is 2 words per packet times 15 slave cores, in total 30 clock cycles.

The *Sparse* benchmark in Table 3.1 reaches this I/O bound. Table 3.1 shows that the TDM periods of some benchmarks are only increased by two clock cycles when VCs for configuration are added. These two slots are the two words of the configuration packet from the master to the node that has the highest incoming bandwidth requirements. The TDM periods of the *All2all* and *FFT-1024* benchmarks are increased considerably, because all cores have high outgoing bandwidths.

3.6.2 Variable-Length Packets

This subsection evaluates the TDM period reduction of the MCSL benchmarks, when allowing variable-length packets. We let the scheduler route fewer packets

Benchmark	w/ VCs for config.	Var.-len. pkt.	
	(cc)	(cc)	(%)
FFT-1024	78	74	5
Fpppp	120	95	21
RS-dec	92	77	16
RS-enc	86	73	15
H264-720p	92	78	15
Robot	171	127	26
Sparse	30	30	0
All2all	75	75	0

Table 3.2: TDM period in clock cycles (cc) and reduction (%) relative to the schedules with VCs for configuration when variable-length packets are allowed.

with more payload words, such that the number of payload bytes during one TDM period is the same as without variable-length packets. Table 3.2 shows the TDM periods with VCs for configuration and packet lengths between 3 and 16 words, which is one header word plus 1 to 15 payload words.

In Table 3.2 we see two benchmarks, Fpppp and Robot, where the reduction is considerable, 21% and 26%, respectively. Looking into the communication patterns of the Fpppp and Robot benchmarks, we see that they have a few cores that are involved in most of the communication. These few cores communicate through VCs that require a high bandwidth, so reducing the header overhead of these VCs causes this more than 20% reduction in TDM period. The variable-length packets reduce the TDM period of most benchmarks, except for the sparse and All2all benchmarks.

3.6.3 Schedule Storage Size

This subsection evaluates the size of the schedules in the schedule table and the size of the DMA controllers in the DMA table. The minimum and maximum number of bytes that it takes to store the schedule in the schedule table of one node in the platform is shown in Table 3.3, for each MCSL benchmark and an All2all schedule. The sum of the maximum schedule table sizes of all the MCSL benchmarks and the All2all schedule is 696 bytes for one node. This is an upper bound on the schedule table size that is required to store all the schedules at the same time. For further studies, we assume that a schedule table of 1 KB is enough to keep all the schedules of most applications in the schedule table at the same time, avoiding the need to transmit a new schedule from the master core to all the slave cores through the NoC.

Benchmark	Sched. tbl. (Byte)		DMA tbl. (Byte)		# VC	
	min	max	min	max	min	max
FFT-1024	52	108	74	152	13	27
Fpppp	56	108	74	147	13	26
RS-dec	24	76	23	102	4	18
RS-enc	20	68	6	90	1	16
H264-720p	20	72	6	90	1	16
Robot	32	84	6	85	1	15
Sparse	8	60	6	85	1	15
All2all	60	120	85	169	15	30

Table 3.3: The minimum and maximum number of bytes of storage in the schedule table and in the DMA table of one node, and the minimum and maximum number of outgoing VCs in one node.

The minimum and maximum number of bytes that are required in the DMA table of one node in the platform to execute the DMA controllers are shown in Table 3.3. The required number of bytes in the DMA table is the number of VCs multiplied by the width of the DMA table. The width of the DMA table mainly depends on the read pointer, the write pointer and the word count. The numbers that are shown in Table 3.3 are for a case where 14 bits are used for the three fields, which enables an SPM of 64 KB to be used, and this is also what the current packet format supports. Each entry in the DMA table represent an outgoing VC. Therefore, the memory requirements for the DMA tables of each schedule can be overlapped by the VCs that persist across reconfigurations between the schedules of an application.

In the original version of Argo and in Argo 2.0, the number of entries in the DMA tables of each node is the same, since it is determined by the application. Therefore, we only compare the number of schedule table entries in each node of the original version of Argo against the number of entries in Argo 2.0, in table 3.4. The average reduction in the schedule table entries of each node is 58 %, this improvement is due to the new and more efficient architecture of the Argo 2.0 NI.

3.6.4 Worst-case Reconfiguration Time

This subsection gives an overview of how to calculate the worst-case reconfiguration time T_{recon} of a new schedule C_{new}. T_{recon} depends only on the currently executing schedule C_{curr}. From Fig. 3.7 we see that:

$$T_{\text{recon}} = 3 \cdot P_{\text{curr}} \tag{3.1}$$

Benchmark	Original (entries)	Argo 2.0 (entries)	Reduction (%)
FFT-1024	21	15	28.6
Fpppp	40	16	60.0
RS-dec	30	8	73.3
RS-enc	28	6	78.6
H264-720p	30	7	76.7
Robot	57	10	82.5
Sparse	9	4	55.6
All2all	18	16	11.1

Table 3.4: The number of schedule table entries in each core of the original Argo version (Original.) and the average number of entries in Argo 2.0.

C_{curr}	T_{recon}
FFT-1024	267
Fpppp	285
RS-dec	279
RS-enc	261
H264-720p	279
Robot	381
Sparse	135
All2all	270

Table 3.5: The worst-case reconfiguration time T_{recon}, starting from each benchmark as the current schedule.

Where, P_{curr} is the TDM period of C_{curr}. We calculate T_{recon} of the MCSL benchmark and an All2all schedule, shown in Table 3.5. An application programmer needs to add the software overhead of setting up DMA transfers and triggering a reconfiguration request to the numbers in the table.

For the benchmarks presented in Table 3.5, the T_{recon} is between 135 and 381 clock cycles, depending on the current benchmark. For the maximum number of entries shown in Table 3.3 and assuming a schedule table of 256 entries (1 KB), it is possible to store of the order of 10 schedules. We believe that in most cases this schedule table size is sufficient to store the schedules associated with all the modes of an application.

The time interval required by Æthereal and dAElite to set up a single VC is respectively 246 and 60 clock cycles [28]. Moreover, these NoCs also need to tear down the unnecessary VCs. Assuming that any two modes differ by more

than a handful of VCs, the reconfiguration time of Argo 2.0 is in general shorter than the reconfiguration time of Æthereal and comparable to or less than that of dAElite.

3.6.5 Worst-case Schedule Transmission Time

This subsection gives an overview of how to calculate the schedule transmission time T_{st} of a new schedule C_{new} that is not stored in the schedule tables of the slave processors. The T_{st} of C_{new} depends on the currently executing schedule C_{curr}. The worst-case analysis of software depends on the processor that executes the software. Therefore, we do not include the software overhead of setting up DMA transfers in the T_{st}. We assume that C_{new} is loaded in the processor local SPM of the reconfiguration master.

A NoC schedule is different in each NI and with the compact schedule representation that we evaluated in subsection 3.6.3, the schedules for each NI might be of different sizes. The T_{st} of transferring C_{new} to the slave NIs is the maximum of the individual worst-case latencies for each slave NI. We calculate the T_{st} as:

$$T_{\text{st}} = \max_{i \in N} \left(L^i_{\text{curr}} + \left\lceil \frac{S^i_{\text{new}} - P^i_{\text{curr}}}{B^i_{\text{curr}}} \right\rceil \cdot L_{\text{curr}} + L^i_{\text{chan}} \right) \tag{3.2}$$

Here i is the slave NI from the set N of nodes in the platform, L^i_{curr} is the worst-case latency of waiting for a time slot to slave i, S^i_{new} is the number of words of C_{new} to be sent to slave i, P^i_{curr} is the number of words that can be sent in one packet towards slave i in C_{curr}, B^i_{curr} is the bandwidth of C_{curr} towards slave i, L_{curr} is the TDM period of C_{curr}, and L^i_{chan} is the NoC latency in clock cycles to slave i.

We apply (3.2) to calculate the worst-case schedule transmission time between the schedules of the MCSL benchmark and an All2all schedule, shown in Table 3.6.

We see that the T_{st} in Table 3.6 is between 519 and 3822 clock cycles. The Sparse benchmark, as C_{curr}, results in the lowest T_{st}, as Sparse has the shortest TDM period, and thus the highest bandwidth to the slaves.

In the rare case that a schedule needs to be transmitted to the slave NIs, our approach is still comparable. The maximum schedule transmission time in Table 3.6 is 3822 clock cycles. For Argo 2.0, this transmission represents the transmission of 255 VCs. In this time interval, Æthereal and dAElite can only set-up 16 and 64 VCs, respectively, this does not include tearing-down VCs.

3.6.6 Hardware Results

This subsection presents the evaluation of the Argo 2.0 FPGA implementation presented here with respect to hardware size and maximum operating fre-

C_{curr} \ C_{new}	FFT-1024	Fpppp	RS-dec	RS-enc	H264-720p	Robot	Sparse	All2all
FFT-1024	–	2010	1418	1270	1341	1560	1122	2229
Fpppp	2577	–	1814	1624	1716	2004	579	2862
RS-dec	2091	2088	–	1318	1398	1626	1164	2316
RS-enc	1983	1980	1396	–	1326	1548	1104	2196
H264-720p	2115	2112	1494	1338	–	1644	1176	2355
Robot	3435	3438	2422	2174	2292	–	1914	3822
Sparse	822	549	579	519	546	639	–	912
All2all	2034	2037	1431	1281	1365	1587	1137	–

Table 3.6: Worst-case schedule transmission time of a new schedule C_{new} expressed in clock cycles. Since this depends on both the current schedule C_{curr} and the new one C_{new}, we show a matrix of the combination of current and new schedules.

quency. All the results presented in this section were produced using Xilinx ISE Design Suite (version 14.7) and targeting the Xilinx Virtex-6 FPGA (model XC6VLX240T-1FFG1156). All the synthesis properties were set to their defaults, except for the synthesis optimization goal which were set to area or speed. The results are expressed in terms of numbers of flip-flops (FFs), 6-input look-up tables (LUTs), and block RAMs (BRAMs).

Table 3.7 shows the comparison of the Argo 2.0 implementation to the TDM-based NoCs aelite and dAElite [28], and to the IDAMC [20] NoC that uses a classic router designed with virtual channel buffers and flow control. The table shows the results of the four designs for one router and one NI and the number of supported TDM slots and connections per node. The published numbers we compare against are available for comparison for a 2-by-2 platform with mesh topologies. From these results we derived the hardware consumption of one 3-ported router and one NI.

The results in Table 3.7 show that overall the Argo 2.0 NoC implementation is smaller than the other NoCs. The results also show that the numbers for the IDAMC are much higher that the aelite, dAElite and Argo 2.0. This is due to its use of virtual channels buffers and the flow control mechanisms.

The results in Table 3.7 shows that the maximum frequency f_{max} of the Argo 2.0 implementation is comparable to the ones of aelite and dAElite for a 3-port router. The f_{max} results for the IDAMC NoC are not available for comparison.

As mentioned, the Argo 2.0 NoC is designed to be used in a domain-specific platform. Therefore, Table 3.8 presents numbers for a network node comprising one 5-ported router and one NI with 256 TDM slots and 64 connections, which we consider reasonable numbers for a larger platform. Moreover, it compares the Argo 2.0 NoC against the original Argo NoC in order to show that our

	Optimized for area			Optimized for speed			
	aelite	dAElite	Argo 2.0	aelite	dAElite	Argo 2.0	IDAMC
Slots	8	8	8	8	8	8	N/A
Conn	1	1	2	1	1	2	8
LUTs	1916	2506	1185	2351	3117	1342	9160
FFs	3861	3081	1021	3960	3243	1047	5462
BRAM	0	0	0	0	0	0	7
f_{max} (MHz)	119	122	125	200	201	204	N/A

Table 3.7: Hardware resources and maximum operating frequency of the Argo 2.0 architecture presented here and three similar designs for one 3-port router and one NI.

	Opt. for area		Opt. for speed	
	Argo	Argo 2.0	Argo	Argo 2.0
LUTs	926	1071	1155	1358
FFs	897	908	923	931
BRAM	4	4	4	4
f_{max} (MHz)	155	166	167	179

Table 3.8: Hardware resources and maximum operating frequency of the Argo 2.0 architecture and the original Argo NoC, for one 5-port router and one NI. The implemented design has 256 slots and 64 connections.

extensions only add a small amount of hardware resources. We re-synthesized the original Argo implementation for the results in the table. For the results in Table 3.8, we used BRAM to implement the tables in the NI.

In terms of f_{max}, the Argo 2.0 5-port router implementation optimized for area is around 33% faster than the 3-port one, since it uses BRAM instead of distributed memory (implemented using FFs), and around 7% faster than the original Argo.

3.6.7 Scalability

The results in the previous subsections are based on a 16 core platform. As the number of cores in the platform increases, we consider the hardware size and the TDM period to evaluate the scalability of the new reconfiguration capability of Argo 2.0. We consider the hardware size of the NoC per core and the extension

of the TDM period due to statically allocating VCs for reconfiguration. As the number of cores increase, the hardware size of one NI and one router increases due to the number of required entries in the schedule table and in the DMA table.

The number of entries that are required in the schedule table is the accumulated number of entries per core. Our reconfiguration approach requires the TDM schedule to include VCs for configuration from the master to all slave cores. The lower bound on the TDM period increases linearly with the number of slave cores, due to the increasing number of VCs for configuration. The lower bound is two clock cycles per slave core, which can be represented by one schedule table entry. The aelite NoC has the same property, as it also allocates VCs for reconfiguration. The dAElite NoC does not allocate VCs for configuration in the TDM schedule, but dAElite uses a separate single-master configuration tree network that causes a minor increase in the hardware size. The single-master configuration tree network results in a fixed, i.e., less flexible, allocation of bandwidth between the configuration and data communication.

As mentioned previously, the number of active entries in the DMA table is the number of outgoing VCs from the processor node. The number of DMA table entries in the Argo 2.0 NI grows in the same way as the VC buffers with credit-based flow control grows in the aelite and dAElite NIs. The hardware size of a DMA table entry is considerably smaller than the size of a VC buffer with credit-based flow control.

3.7 Conclusion

This paper presented an area-efficient time-division multiplexing network-on-chip that supports reconfiguration for mode changes. The NoC addresses hard real-time systems and provides guaranteed-service VCs between processors. The NI provides reconfiguration capabilities of end-to-end VCs to support mode changes at the application level. For the set of benchmarks used for evaluation, we showed that the TDM period overhead of statically allocating VCs for configuration was on average 10%. Furthermore, we showed that our compact schedule representation reduces the memory requirements by more than 50% on average.

We evaluated an implementation of the proposed architecture in terms of hardware cost and worst-case reconfiguration time. The results show that the proposed architecture is less than half the size of NoCs with similar functionality and that the worst-case reconfiguration time is comparable to those NoCs. If the new schedule is already loaded in the schedule table, the worst-case reconfiguration time is significantly shorter.

Acknowledgment

The work presented in this paper was funded by the Danish Council for Independent Research | Technology and Production Sciences under the project RTEMP[1], contract no. 12-127600.

Source Access

The presented work is open source and can be downloaded from GitHub and built under Ubuntu as described in the Patmos reference handbook [24, Chap. 6].

Bibliography

[1] L. Benini, E. Flamand, D. Fuin, and D. Melpignano. P2012: Building an ecosystem for a scalable, modular and high-efficiency embedded computing accelerator. In *Proc. Design, Automation and Test in Europe (DATE)*, pages 983–987, 2012.

[2] D. Berozzi. Network interface architecture and design issues. In G. DeMicheli and L. Benini, editors, *Networks on Chips*, chapter 6, pages 203–284. Morgan Kaufmann Publishers, 2006.

[3] D. Bertozzi, A. Jalabert, S. Murali, R. Tamhankar, S. Stergiou, L. Benini, and G. D. Micheli. NoC synthesis flow for customized domain specific multiprocessor systems-on-chip. *IEEE Trans. Parallel and Distributed Systems*, 16(2):113–129, 2006.

[4] T. Bjerregaard and J. Sparsø. A router architecture for connection-oriented service guarantees in the MANGO clockless network-on-chip. In *Proc. Design, Automation and Test in Europe (DATE)*, pages 1226–1231. IEEE Computer Society Press, 2005.

[5] A. Burns and A. Wellings. *Real-Time Systems and Programming Languages: Ada 95, Real-Time Java and Real-Time POSIX*. Addison-Wesley, 2001. ISBN 0201729881, 9780201729887.

[6] G. Chen, M. A. Anders, H. Kaul, S. K. Satpathy, S. K. Mathew, S. K. Hsu, A. Agarwal, R. K. Krishnamurthy, V. De, and S. Borkar. A 340 mV-to-0.9 V 20.2 Tb/s Source-Synchronous Hybrid Packet/Circuit-Switched 16 x 16 Network-on-Chip in 22 nm Tri-Gate CMOS. *IEEE Journal of Solid-State Circuits*, 50(1):59–67, 2015.

[7] F. Clermidy, C. Bernard, R. Lemaire, J. Martin, I. Miro-Panades, Y. Thonnart, P. Vivet, and N. Wehn. A 477mW NoC-based digital baseband

[1] `http://rtemp.compute.dtu.dk`

for MIMO 4G SDR. In *Proc. IEEE Intl. Solid-State Circuits Conference (ISSCC)*, pages 278–279, 2010. doi: 10.1109/ISSCC.2010.5433920.

[8] R. L. Cruz. A calculus for network delay. I. Network elements in isolation. *IEEE Transactions on Information Theory*, 37(1):114–131, Jan 1991. ISSN 0018-9448. doi: 10.1109/18.61110.

[9] J. Diemer and R. Ernst. Back suction: Service guarantees for latency-sensitive on-chip networks. In *ACM/IEEE International Symposium on Networks-on-Chip (NOCS)*, pages 155–162, May 2010. doi: 10.1109/NOCS.2010.38.

[10] B. Dupont de Dinechin, Y. Durand, D. van Amstel, and A. Ghiti. Guaranteed services of the NoC of a manycore processor. In *Proc. Intl. Workshop on Network on Chip Architectures (NoCArc)*, pages 11–16, New York, NY, USA, Dec. 2014. ACM.

[11] B. Dupont de Dinechin, D. van Amstel, M. Poulhiès, and G. Lager. Time-critical computing on a single-chip massively parallel processor. In *Proc. Design, Automation and Test in Europe (DATE)*, pages 97:1–97:6, 2014.

[12] K. Goossens and A. Hansson. The aethereal network on chip after ten years: Goals, evolution, lessons, and future. In *Proc. ACM/IEEE Design Automation Conference (DAC)*, pages 306 –311, June 2010.

[13] K. Goossens, J. Dielissen, and A. Radulescu. AEthereal network on chip: concepts, architectures, and implementations. *IEEE Design Test of Computers*, 22(5):414–421, Sept 2005. ISSN 0740-7475. doi: 10.1109/MDT.2005.99.

[14] A. Hansson, K. Goossens, M. Bekooij, and J. Huisken. Compsoc: A template for composable and predictable multi-processor system on chips. *ACM Transactions on Design Automation of Electronic Systems (TODAES)*, 14 (1):2, 2009.

[15] E. Kasapaki, J. Sparsø, R. B. Sørensen, and K. Goossens. Router designs for an asynchronous time-division-multiplexed network-on-chip. In *in Proc. Euromicro Conference on Digital System Design (DSD)*, pages 319–326. IEEE, 2013.

[16] E. Kasapaki, M. Schoeberl, R. B. Sørensen, C. T. Müller, K. Goossens, and J. Sparsø. Argo: A Real-Time Network-on-Chip Architecture with an Efficient GALS Implementation. *IEEE Transactions on VLSI Systems*, 24 (2):479–492, 2015.

[17] M. Kistler, M. Perrone, and F. Petrini. Cell multiprocessor communication network: Built for speed. *Micro, IEEE*, 26:10–25, 2006. URL `http://ieeexplore.ieee.org/iel5/40/34602/01650177.pdf`.

[18] I. Kotleas, D. Humphreys, R. Sørensen, E. Kasapaki, F. Brandnery, and J. Sparsø. A loosely synchronizing asynchronous router for tdm-scheduled nocs. pages 151–158, 2014.

[19] W. Liu, J. Xu, X. Wu, Y. Ye, X. Wang, W. Zhang, M. Nikdast, and Z. Wang. A NoC traffic suite based on real applications. In *VLSI (ISVLSI), 2011 IEEE Computer Society Annual Symposium on*, pages 66 –71, July 2011.

[20] B. Motruk, J. Diemer, R. Buchty, R. Ernst, and M. Berekovic. IDAMC: A many-core platform with run-time monitoring for mixed-criticality. In *Proc. IEEE Intl. Symposium on High-Assurance Systems Engineering (HASE)*, pages 24–31, 2012. doi: 10.1109/HASE.2012.19.

[21] S. Murali, G. D. Micheli, A. Jalabert, and L. Benini. XpipesCompiler: A tool for instantiating application specific networks-on-chip. In *Proc. Design, Automation and Test in Europe (DATE)*, pages 884–889. IEEE Computer Society Press, 2004.

[22] A. Olofsson, T. Nordström, and Z. ul Abdin. Kickstarting high-performance energy-efficient manycore architectures with Epiphany. In M. B. Matthews, editor, *in Proc. Asilomar Conference on Signals, Systems and Computers*, pages 1719–1726. IEEE, 2014. ISBN 978-1-4799-8297-4. URL `http://ieeexplore.ieee.org/xpl/mostRecentIssue.jsp?punumber=7066991`.

[23] A. Radulescu, J. Dielissen, S. Pestana, O. Gangwal, E. Rijpkema, P. Wielage, and K. Goossens. An efficient on-chip NI offering guaranteed services, shared-memory abstraction, and flexible network configuration. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 24(1):4–17, 2005. ISSN 02780070. doi: 10.1109/TCAD.2004.839493.

[24] M. Schoeberl, F. Brandner, S. Hepp, W. Puffitsch, and D. Prokesch. Patmos reference handbook. Technical report, 2014. URL `http://patmos.compute.dtu.dk/patmos_handbook.pdf`.

[25] R. B. Sørensen, J. Sparsø, M. R. Pedersen, and J. Højgaard. A metaheuristic scheduler for time division multiplexed networks-on-chip. In *Proc. IEEE/IFIP Workshop on Software Technologies for Future Embedded and Ubiquitous Systems (SEUS)*, pages 309–316, 2014.

[26] R. B. Sørensen, L. Pezzarossa, and J. Sparsø. An area-efficient TDM NoC supporting reconfiguration for mode changes. In *Proc. of ACM/IEEE International Symposium on Networks-on-Chip (NOCS)*. IEEE, 2016. (Attached at the end of this manuscript).

[27] J. Sparsø, E. Kasapaki, and M. Schoeberl. An area-efficient network interface for a TDM-based network-on-chip. In *Proc. Design, Automation and Test in Europe (DATE)*, pages 1044–1047, 2013.

[28] R. A. Stefan, A. Molnos, and K. Goossens. dAElite: A TDM NoC Supporting QoS, Multicast, and Fast Connection Set-Up. *IEEE Transactions on Computers*, 63(3):583–594, 2014. ISSN 00189340, 15579956. doi: 10.1109/TC.2012.117.

Chapter 4

A Metaheuristic Scheduler For Time Division Multiplexed Networks-on-Chip

This chapter was previously published as: © 2014 IEEE. Reprinted, with permission, from Rasmus Bo Sørensen, Jens Sparsø, Mark Ruvald Pedersen, and Jaspur Højgaard,
"A Metaheuristic Scheduler For Time Division Multiplexed Networks-on-Chip",
10th Workshop on Software Technologies for Future Embedded and Ubiquitous Systems (SEUS), pages 309–316, 2014.

Abstract

This paper presents a metaheuristic scheduler for inter-processor communication in multi-processor platforms using time division multiplexed (TDM) networks on chip (NOC). Compared to previous works, the scheduler handles a broader and more general class of platforms.
Another contribution, which has significant practical implications, is the minimization of the TDM schedule period by over-provisioning bandwidth to connections with the smallest bandwidth requirements. Our results show that this is possible with only negligible impact on the schedule period.
We evaluate the scheduler with seven different applications from the MCSL NOC benchmark suite. In the special case of all-to-all communication with equal bandwidths on all communication channels, we obtain schedules with a shorter period than reported in previous work.

4.1 Introduction

In this paper we address the scheduling of data-traffic in a TDM-based NOC that supports message passing across virtual end-to-end circuits in a multi-processor platform. There are two rather different variations of mapping applications to hardware platforms. The first variation maps the application onto a given hard real-time application-independent platform. The second variation synthesizes a specific NOC platform for the given application, including a TDM schedule.

The increasing cost of designing and fabricating integrated circuits and the limited fabrication volume of most hard real-time systems speaks strongly in favor of application-independent platforms. This is what we address in this paper. The context for our work is the T-CREST project [6] in which we are developing an application-independent multi-processor platform designed for hard real-time systems, and for which the scheduler described in this paper is being used. Details of the hardware implementation may be found in [18, 8, 14].

The mapping of an application onto an application-independent NOC-based multi-processor platform is generally understood to include the steps illustrated in Figure 4.1 [12, 10]. An application is modeled as a *task graph* where nodes represent tasks and edges represent communication channels (end-to-end circuits). The first step is to assign tasks to processors and as part of this to decide which tasks will share a processor. The result of this is a *core communication graph* where the nodes represent processors and the edges represent communication flows and their required bandwidth between the processors. The second step is the binding of processors to specific processor cores in the platform. This usually aims at minimizing the total number of router-to-router hops for traffic. The first and second steps are not specific for TDM-based NOCs and they are well studied in the literature. Early works include [12]. The third step, and the topic of this paper, is to generate the TDM schedule for the NOC.

An important challenge is to make a generic and parameterized specification of the NOC-based platform allowing the scheduler to target a large class of different multi-processor platforms using TDM-based NOCs. These parameters include the topology of the NOC (regular as well as irregular topologies), the packet length, and the pipeline depth of the routers and links. The scheduling problem can be modeled as a fixed-flow, minimum-time integer multi-commodity flow problem that is known to be NP-complete [4]. In this case each communication channel is modeled as a commodity.

The paper makes contributions in two areas:

- An open source scheduler that is more general in terms of parameterization and produces better results than previously published schedulers.

- The novel idea of minimizing the number of slots in a period of the generated TDM schedule.

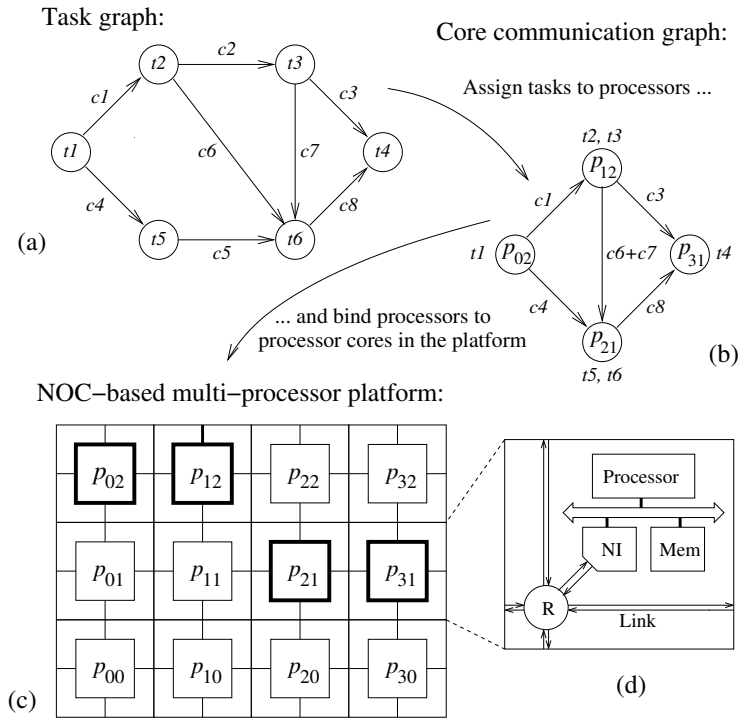


Figure 4.1: Mapping of an application onto a multi-processor platform: (a) Task graph for application, (b) core communication graph, (c) multi-processor platform, and (d) details of a node in the platform (router (R), links, network interface (NI), processor and local memory).

The goal is to reduce the size of the potentially very large schedule-tables such that they fit into the memory structures that are implemented in the hardware platform. Our results show that it is possible to compress the schedules to around 100 slots for actual benchmarks, with only a negligible increase of the frequency of the TDM clock.

The paper is organized as follows. In Section 4.2 we review related work. In Section 4.3 we discuss and identify the details of the scheduling problem. Following this, Section 4.4 presents the design and implementation of the scheduler. A description of the benchmarks used in our experiments is given in Section 4.5. Results from a range of benchmarks are presented in Section 4.6 and discussed in Section 4.7. Finally, Section 4.8 concludes the paper.

4.2 Related work

The UMARS scheduler described in [7] can generate TDM schedules and an application-specific Æthereal platform. This allows the scheduler to modify the topology of the NOC and the pipeline depth of the NOC links to obtain feasible schedules that satisfy the bandwidth requirements. The UMARS scheduler operates in two phases: path allocation and TDM-slot allocation. The path allocation phase is an all-pair shortest-path search for the possible routes for each communication channel. The TDM-slot allocation is a collision free mapping of the allocated paths to TDM slots. The UMARS scheduler works at the level of 3-word flits/packets and pipelining of routers and links is done in multiples of three. Our work addresses the communication mapping of an application onto an application-independent platform.

Nostrum is another NOC that is based on TDM scheduling [11]. The scheduler supports regular topologies (mesh, torus, etc.) and it has the same two phases (path selection and slot allocation) as the UMARS scheduler. Compared to a conventional packet-switched NOC, Nostrum is a bit more elaborated. It uses temporal-disjoint networks and looped containers (like multiple slotted rings on top of an underlying mesh-style topology). The number of virtual circuits through a router is limited to the number of temporally disjoint networks.

The mapping and scheduling of fully connected core communication graphs that offer virtual circuits with identical bandwidth between all pairs of processor nodes onto platforms with regular NOC topologies (mesh, torus, tree, etc.) is studied in [15]. The paper provides a number of theoretical lower bounds on the schedule length and it presents an ILP-based scheduler that produces optimal schedules, in terms of schedule length. A very high run-time limits the scheduler to platforms with a small or modest number of nodes – for a platform with 25 nodes the run-time is reported to be 2 weeks. A heuristic solution to the problem is studied in [2]. A unique aspect of this work is that the routing is identical in all routers. This may allow sharing of routing tables in routers in technologies where this is possible (e.g. FPGAs).

Our scheduler is different from the above work, because it is not limited to symmetric all-to-all core communication graphs, it targets an application-independent platform, and it accepts a highly parameterized specification of the TDM-based NOC in the platform (NOC-topology, pipeline depth of routers and links, different packet lengths on different channels in the core communication graph, etc.). In conclusion our scheduler is far more generic than previously published work and it reveals some interesting insight into the period of the TDM schedules and ways to reduce the period.

4.3 The Scheduling Problem

The task of the scheduler is to schedule and route the communication channels of the core communication graph onto the TDM-based NOC in a multi-processor platform. The nodes in a core communication graph are processor cores and the edges are communication channels annotated with bandwidth requirements. The NOC must be configured to implement the communication channels.

Definition 4.1 An application has the directed core communication graph $A(P, C)$, where P is the set of processors and C is the set of communication channels.

Definition 4.2 A communication channel $c \in C$ is the triple (p_{src}, p_{dest}, b), where $p_{src} \in P$ is the sending processor, $p_{dest} \in P$ is the receiving processor and $b \in \mathbb{R}$ is the required bandwidth in MB/s.

The aim of the scheduling is to avoid situations where multiple packets compete for the same resource, i.e. a link in the NOC or an output port of the router. This requires a common time reference (e.g. a clock signal) that defines the time slots that are the basis for the scheduling. In the following we call this the TDM clock. A packet consists of a sequence of data-words that is sent in a corresponding sequence of TDM-clock cycles. The routers and links in the NOC are typically pipelined, which has to be considered when scheduling the traffic.

The TDM clock should be seen as variable parameter that can be set for a given application. In most situations, the bandwidth required by the application does not need the NOC to run at its maximum clock frequency. This allows the use of a TDM clock with a lower frequency.

Input to the scheduler is a specification of the application and a specification of the platform. The application is specified by a core communication graph as explained in Section 4.1 and Figure 4.1.

A platform specification describes the platform on which to route the communication from the core communication graph. The platform specification describes routes and links, as explained in Section 4.1, the routers and links have multiple parameters. In the following definitions we assume that pipeline depths for routers and links are 1 and 0, respectively, in order to keep the definitions simple. It is also possible to specify the packet size for each individual channel in the core communication graph, but in most cases the entire NOC will use only one packet size.

Definition 4.3 A platform specification is the directed graph $T(N, L)$, where N is the set of nodes in the platform and L is the set of directed physical links connecting two unique nodes. A node $n \in N$ consists of a router connected to a local processor. A directed physical link $l \in L$ is the tuple (n_{src}, n_{dest}), where $src, dest \in \mathbb{N}^2$, i.e., two-dimensional Cartesian coordinates.

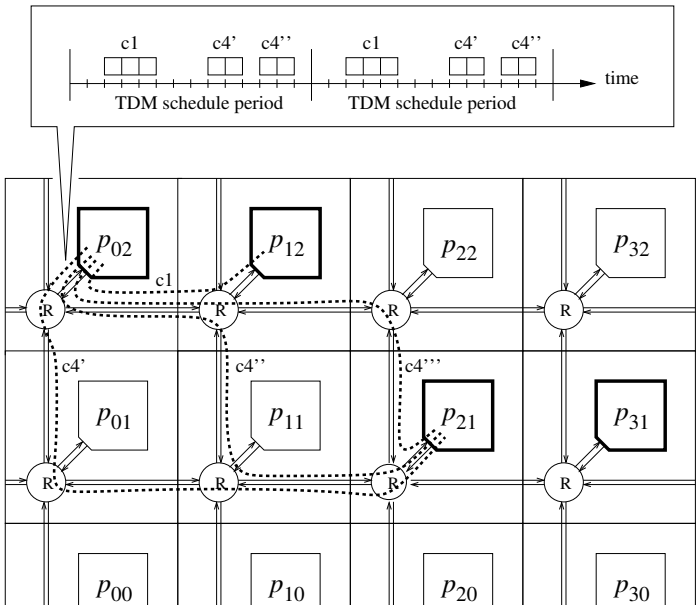


Figure 4.2: Possible routing and scheduling of traffic out of processor core p_{02} (channels c1 and c4 from figure 4.1(b). Several or all of the paths marked c4', c4" and c4"' may be used to implement channel c4. In the schedule shown, only paths c4' and c4" are used.

Given a core communication graph and a platform specification, the task of the scheduler is to determine the routing and scheduling of the data traffic in the NOC. Figure 4.2 shows a close-up of Figure 4.1(c) and illustrates the routing and scheduling of traffic out of processor core p_{02}, i.e., channels c1 and c4 in the core communication graph shown in Figure 4.1(b). The scheduler allows a channel in the core communication graph to be provided by multiple communication paths through the NOC. To ensure that packets arrive in order, the scheduler allows only shortest-path communication channels. For channel c1 there is only one option, as processor cores p_{02} and p_{12} are direct neighbors (connected by one link between routers in tiles 02 and 12). For channel c4, three different paths marked c4', c4' and c4"' may be used. This assumes that the total pipeline depth from p_{02} to p_{12} is the same along all three paths (to ensure in-order delivery of packets). The ports connecting p_{02} and p_{21} to their respective router must be able to carry all the data traffic corresponding to channels c1 and c4.

A communication path is the route of one packet traversing the NOC from its source to its destination. The communication path consists of a sequence

of neighboring scheduled links in consecutive time slots. A scheduled link is a physical link together with a time slot.

Definition 4.4 A scheduled link $s \in S$ is the pair (l, t), where $l \in L$ is the physical link considered at time slot $t \in \mathbb{N}_0$.

Definition 4.5 A communication path is the vector $\vec{\phi} = \langle s_0, s_1, \ldots, s_{\text{last}} \rangle \in \Phi$ of scheduled links $s \in S$ with $\vec{\phi}_{[i]}.t + 1 = \vec{\phi}_{[i+1]}.t$ and $\vec{\phi}_{[i]}.l.p_{dest} = \vec{\phi}_{[i+1]}.l.p_{src}$. Φ is the set of all possible communication paths.

When the scheduler schedules the communication channels, it needs to know how many packets it needs to schedule in one TDM period. The bandwidth of the communication channels is given in MB/s, this bandwidth needs to be normalized to a number of packets per period. Because we want to minimize the schedule period, we cannot directly convert between MB/s and the number of packets per TDM period. Therefore, we normalize the bandwidth of each communication channel to the bandwidth of the communication channel with the smallest bandwidth requirement. This bandwidth is rounded up to the nearest integer; this (dimensionless) bandwidth is then the required number of packets in one TDM period.

Definition 4.6 The normalization of the bandwidth of a communication channel is given by the norm function.

$$\text{norm} : b \mapsto \left\lceil \frac{b}{\min\limits_{c_m \in C} c_m.b} \right\rceil$$

A valid schedule is a specific set of communication paths that satisfy the normalized bandwidth requirements of the core communication graph. In a valid schedule, no link must be part of two different communication paths in the same time slot. The objective of the scheduler is to minimize the TDM period. This minimization may be exploited to decrease latency (and increase bandwidth), or to lower the frequency of the TDM clock (while preserving the bandwidth).

Definition 4.7 A schedule is a specific set of paths $\Phi^* \subset \mathcal{P}(\Phi)$, where $\mathcal{P}(\cdot)$ is the powerset. The schedule period of a schedule is the function $\text{period} : \Phi^* \mapsto \max\limits_{\vec{\phi} \in \Phi^*} \{\vec{\phi}_{[\text{last}]}.t\}$.

Definition 4.8 The scheduler $\text{sched} : (A, T) \to \Phi^*$ maps the channels of A, given the platform T, to a specific set of communication paths Φ^*, i.e., a valid schedule.

Constraint: No overlapping paths $\vec{\phi}_a \neq \vec{\phi}_b : \forall\limits_{i,j} \vec{\phi}_{a[i]} \neq \vec{\phi}_{b[j]}$

Objective: *Minimize* period(Φ^*), where Φ^* is the produced schedule from specific A^*, T^*.

In the generated schedule, the channel with the smallest bandwidth requirement is scheduled to send exactly one packet and the scheduler aims at generating a schedule with the shortest possible period that satisfies the normalized bandwidth requirements. Based on this schedule period, the minimum frequency of the TDM clock that satisfies the absolute bandwidth requirements is determined.

The normalization of bandwidth requirements *may* result in very long schedules. As the TDM schedule table for each node has to be implemented in hardware in the NI, it is desirable to limit the number of table entries and therefore to derive schedules with a modest period. Based on previously published hardware designs, we consider 64-128 entries as realistic and 256 entries on the high side. By assigning more bandwidth to the channel with the smallest required bandwidth, the normalized bandwidth of all other channels is reduced, and this results in a reduced TDM-schedule period. During a physical time window a channel of cause needs the same number of slots and the compressed TDM schedule is simply repeated more times. In many cases the TDM-schedule period is reduced by the same factor as the normalized bandwidths are reduced, and this means that the frequency of the TDM clock is the same. When aggressively compressing the TDM schedule, the NOC may start to saturate, and the schedule period is no longer reduced proportional to the normalized bandwidths. This may be compensated for by increasing the frequency of the TDM clock. This schedule compression is an important contribution of the paper.

As the platform is given, there is no guarantee that the required absolute bandwidths (for example MB/s) can be supported by the platform. To accept a produced schedule, we need to verify that it provides enough bandwidth on the specified platform. If the inequality in equation 4.1 holds, the produced schedule provides more than the required bandwidth from the core communication graph.

$$B_{req} < \frac{B_{sched}}{\text{period}} \cdot D_p \cdot f_{max} \tag{4.1}$$

In equation 4.1 B_{req} is the maximum bandwidth of any channel in the core communication graph in MB/s, and B_{sched} is the number of time slots allocated to the communication channel with the largest bandwidth requirement in one TDM-schedule period. period is number of slots in a TDM period, f_{max} is the maximum operating frequency of the NOC, and D_p is the number of bytes that can be transferred in one time slot.

The inequality only needs to be verified for the communication channel with the largest required bandwidth, because the remaining communication channels are assigned more than the required bandwidth.

4.4 The metaheuristic scheduler

Our metaheuristic scheduler produces a schedule that satisfies the given normalized bandwidth requirements while minimizing the TDM schedule period. It implements two different metaheuristic algorithms. In this section we give a brief outline of the metaheuristic algorithms that we have used and their implementation.

4.4.1 Metaheuristics

A metaheuristic is a high-level optimization strategy that can be used to explore large search spaces. The non-problem-specific metaheuristics guide the search process and the search process is usually non-deterministic [1]. Metaheuristics are used to find good, but not guaranteed optimal, solutions to NP-hard problems.

The two metaheuristics we have implemented are Greedy Randomized Adaptive Search Procedure (GRASP) [5] and Adaptive Large Neighborhood Search (ALNS) [13]. GRASP and ALNS work well for problems with no clear sense of direction, as opposed to the metaheuristic TABU search [3, Chapter 6], which saves the path in the solution space that it has already searched to avoid going back to an already visited solution.

The GRASP pseudo code is shown in Algorithm 1. GRASP creates a greedy randomized initial solution and tries to improve it through a local search, until it finds a local optimum. This local search is performed by selecting an operator from the operator table. Each entry in the operator table is an operator and the probability of the given operator being selected. The probabilities in the operator table are updated after each of the iterations, depending on the results of the performed local search. This process of creating an initial solution and improving it is then repeated for a given amount of time. In each of the iterations, the best solution is updated if the current solution is better.

The ALNS pseudo code is shown in Algorithm 2. ALNS creates an initial solution that satisfies the normalized core communication graph. Then part of the solution is destroyed and repaired and this process is repeated. Section 4.4.2 describes the methods for generating initial solution that we have implemented and tested. In the destroy function, the operator is chosen probabilistically, and the probabilities are updated according to the improvements of the different operators in each of the iterations. The operators select which paths to destroy; after the paths are destroyed the same paths are repaired in a random greedy fashion. This is done for a given amount of time, and the globally best solution is saved.

To allow ALNS to move away from local minima, we want the probability of choosing a given operator to converge slowly. This slow convergence is similar

Algorithm 1 GRASP(A, T) – Pseudo code for the GRASP metaheuristic.

Require:
A: the normalized core communication graph
T: the platform specification

```
1: Best ← initialSolution(A, T)
2: operator ← OperatorTable.select()
3: Best.localSearch(operator)
4: while Time left do
5:    Solution ← initialSolution(A, T)
6:    operator ← OperatorTable.select()
7:    Solution.localSearch(operator)
8:    if eval(Solution) better eval(Best) then
9:       Best ← Solution
10:   OperatorTable.update()
11: return Best
```

to setting a higher temperature in simulated annealing [3, Chapter 7]. The operators are explained in more detail in Section 4.4.3.

The metaheuristic algorithms continuously try to minimize the number of slots in the TDM schedule; therefore a user should allow the scheduler to run for as long as can be afforded. Since the schedule is generated at compile time, it is affordable to let it run for several hours.

4.4.2 Generating initial solutions for GRASP and ALNS

The initial solutions are built to satisfy the bandwidth requirements of the normalized core communication graph; they are built using a greedy algorithm with an adjustable degree of randomness. We have applied randomization for the two types of routing decisions in the algorithm. The first is the decision of which output port to choose when a communication path is routed. The second is the order in which the communication paths are routed. In the deterministic case, the algorithm sorts the set of unscheduled paths by the length from its source to its destination. The sorted set of communication paths is then placed in the schedule one at a time, always picking the longest remaining channel. In any case, a communication path is routed in the earliest possible time slot. We have implemented the algorithm with the following combinations of randomization:

1. Deterministic, no randomization.

2. Randomization of choosing the next output port.

3. Randomization of both the next output port and order of paths.

Algorithm 2 ALNS(A, T) – Pseudo code for the ALNS metaheuristic.

Require:
A: the normalized core communication graph
T: the platform specification

```
1: Current ← initialSolution(A, T)
2: Best ← Current
3: while Time left do
4:    operator ← OperatorTable.select()
5:    Current.destroy(Current)
6:    Current.repair(Current)
7:    if eval(Current) better eval(Best) then
8:       Best ← Current
9:    OperatorTable.Update()
10: return Best
```

4. (Only for GRASP) Takes parameter β, the percentage of paths to be swapped in the sorted set of unscheduled paths. If $\beta = 0$, the behavior is equal to the second combination of randomization. If $\beta = 1$ the behavior is equal to the third combination of randomization.

Good values for beta have been found by running the algorithm on many different problems of different sizes with a wide range of β values. For mesh topologies, we found 0.2 to be a good value and for bi-torus we found 0.02 to be a good value.

4.4.3 Operators

In this subsection we discuss which changes to a solution can lead to optimizations, and we select which operations to implement and use in the scheduler. In order to decrease the period of an existing solution, the end of the schedule should be moved backwards. Intuition says that an existing solution generated by the scheduler is more dense in the beginning than in the end, therefore there is more freedom to reroute paths in the end of the schedule. After each of the optimization iterations, all communication paths are routed completely. The quality of a solution is measured by the period of the TDM schedule.

Definition 4.9 The dominating paths are the set
$\mathrm{D} = \{\vec{\phi} \in \Phi^* : \vec{\phi}_{[last]}.t = \mathrm{period}(\Phi^*)\}$

The dominating paths are the direct cause preventing a shorter schedule. Removing complete communication paths and rerouting them one at a time will not reduce the number of slots in the TDM schedule. Removing a collection of communication paths that prevent each other from being routed earlier might

lead to a shorter schedule period when they are rerouted randomly. To make adaptive decisions on which operators to choose, we need a set of diverse operators to choose from. We have implemented the operators: *Dominating paths*, *Dominating rectangle*, *Late paths* and *Random*.

Definition 4.10 Three basic selection functions:
links : $\vec{\phi} \mapsto \{0 \leq i \leq \vec{\phi}_{[\text{last}]} : \vec{\phi}_{[i]}.l\}$
touches : $L^*, \Phi^* \mapsto \{\vec{\phi} \in \Phi^* : \text{links}(\vec{\phi}) \cap L^* \neq \emptyset\}$
rect : $\vec{\phi} \to \mathcal{P}(L)$ returns the links within the bounding box spanned by $\vec{\phi}$.

The dominating paths operator selects the dominating paths and the paths that are routed on the same physical links as the dominating paths, no matter which time slot they are routed in.

Definition 4.11 The dominating paths operator selects the set DP of paths to reroute, where

$$\text{DP} = \text{touches}\left(\bigcup_{\vec{\phi} \in \text{D}} \text{links}(\vec{\phi}), \Phi^*\right)$$

The dominating rectangle operator selects all paths that are routed on the physical links in the bounding box of each dominating path, no matter which time slot they are routed in.

Definition 4.12 The dominating rectangle operator selects the set DR of paths to reroute, where

$$\text{DR} = \text{touches}\left(\bigcup_{\vec{\phi} \in \text{D}} \text{rect}(\vec{\phi}), \Phi^*\right)$$

The late paths operator selects the paths that end in the last time slot (the dominating paths) and the paths that end in the second-last time slot.

Definition 4.13 The late paths operator selects the set
$\text{DL} = \{\vec{\phi} \in \Phi^* : \vec{\phi}_{[last]}.t \geq \text{period}(\Phi^*) - 1\}$

The Random operator selects a random-sized set of randomly selected paths. At least two paths are always selected and up to 10 % of all existing paths can be selected.

4.4.4 Implementation

Our scheduler is written in C++11, using BOOST 1.49[16] and pugixml 1.0[1] as 3rd party libraries. The source code of our scheduler can be downloaded at `https://github.com/t-crest/poseidon.git`. The scheduler reads an input XML file describing the core communication graph of the application and the platform specification. The communication is then scheduled and the resulting schedule is written to an output XML file.

4.5 Benchmarks

For the benchmarks in this paper, we assume that our platform is a square bitorus. In addition we assume that the pipeline depth of all routers is one, that the pipeline depth of all links is zero, and that the packet size is one data word. Additional benchmarks can be seen in the technical report [17].

We will experiment with two different types of benchmarks. The first is the special case of all-to-all communication where core communication graphs are fully connected graphs with a bandwidth of one on each channel. The second type is the more general case of application-specific schedules from the MCSL benchmarks suite[9]. In all our experiments, we create a core communication graph and a platform specification. For the platform specification we only change the network size.

In the MCSL benchmark suite the tasks are already mapped onto processors, so the benchmarks are basically core communication graphs. As some processors execute more tasks, and as these tasks may have different communication behaviors, we associate the maximum data rate as the required bandwidth for the different channels in the core communication graph. The normalized core communication graph is found as described in Section 4.3.

If the produced schedule is longer than the number of time slots supported by the hardware platform, the schedule needs to be shortened/compressed. One way to achieve this is to normalize the bandwidths with a larger constant than the smallest bandwidth. Another normalization function with a normalization factor as a parameter can be seen in Equation 4.2. The normalization function normf takes the bandwidth and the new normalization factor as input parameters.

$$\text{normf} : b, \sigma \mapsto \left\lceil \frac{b}{\sigma \min\limits_{c_m \in C} c_m.b} \right\rceil \tag{4.2}$$

[1] Source code can be downloaded at `http://pugixml.googlecode.com/files/pugixml-1.0.zip`

Table 4.1: A comparison of the schedule length for the all-to-all case. The schedule lengths are expressed in TDM slots, and the numbers in bold are the best results for the given topology and network size.

Bi-torus	Previous work			This work		
Size	Opt. [15]	Lower bound [15]	Sym. [2]	GREEDY	ALNS	GRASP
3×3	10	8	11	12	**10**	**10**
4×4	18	15	20	21	**19**	**19**
5×5	28	24	**28**	32	30	30
6×6	–	35	–	45	45	**43**
7×7	–	48	–	64	63	**61**
8×8	–	64	88	87	87	**85**
9×9	–	90	–	**113**	**113**	**113**
10×10	–	125	158	154	154	**151**
15×15	–	420	481	**471**	**471**	477

Where, σ is the normalization factor. Normalizing with a large σ degrades the relationship between bandwidths, because the minimum bandwidth is always 1. As σ increases at some point the overall performance drops. Taken to the extreme, the shortest possible schedule is when we normalize with a factor equal to the bandwidth of the largest communication channel. This results in a normalized core communication graph where all channels have a required bandwidth of one packet per TDM period. The TDM period needs to be repeated σ times to provide the same amount of data to be transferred as the uncompressed schedule.

4.6 Results

The scheduler always produces a solution that satisfies the bandwidth requirements of the normalized core communication graph, and the goal of the scheduler is to minimize the schedule period. This again minimizes the frequency of the TDM clock at which the absolute required bandwidths are met. Below we present results for the all-to-all communication and for the core communication graphs derived from the MCSL benchmark applications.

4.6.1 All-to-All Communication

To evaluate the performance of our scheduler in the special case of all-to-all communication, we schedule the communication patterns for different network sizes on both mesh and bi-torus topologies. Table 4.1 shows the results and compares against the results of the heuristic scheduler in [2] and against the optimal

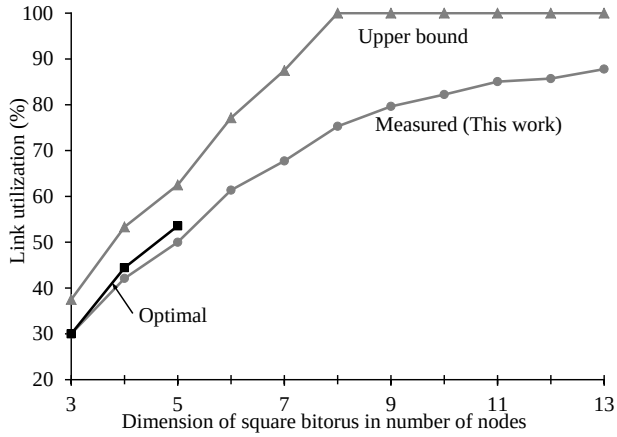


Figure 4.3: The link utilization of a greedy all-to-all schedule on a network with a bi-torus topology. The optimal link utilization is calculated from[15], along with the upper bound on the link utilization that is derived from the lower bound of the schedule period.

results and theoretical lower bounds given in [15]. All our results have been obtained by running our metaheuristic scheduler for two hours on a computer with an i7-3630QM (4 cores @ 2.4 GHz) with 16 GB of memory.

As seen in Table 4.1 our scheduler produces better results than the symmetric scheduler [2] in all cases except the 5×5 bi-torus. This conforms to the fact that the symmetric scheduler is restricted to produce solutions where all routers execute the same schedule, whereas our scheduler has more freedom and hence is able to find solutions with a shorter schedule period.

For the bi-torus our schedules are approx. 30 % longer than the analytical lower bound from [15]. As seen in Table 4.1, the optimal schedule period for a 3×3 bi-torus network is 10, and both our ALNS and GRASP schedulers are able to find schedules with this period. For the other cases for which optimal schedules are known, our scheduler finds solutions whose schedule periods are only slightly larger.

Comparing the GREEDY solutions with the metaheuristic solutions (ALNS and GRASL) produced in Table 4.1, we see that the metaheuristic algorithms produce better results in most cases. For the large network sizes, the improvement by the metaheuristics diminishes. There are several reasons for this. Firstly the link utilization increases with the number of nodes as seen in Figure 4.3 for the bi-torus and the greedy scheduler. The link utilization increases because the average communication channel length grows with the network size. This limits the ability to reroute paths.

Table 4.2: The schedule period (measured in TDM slots) for applications from the MCSL benchmark suite [9] mapped to different network sizes. Numbers marked with a *and a †are generated by ALNS and GRASP, respectively. For unmarked numbers ALNS and GRASP generated schedules of the same length.

Bi-torous network size	Reed Sol. enc.	Reed Sol. Dec.	Dyn. Robot Ctl.	SPEC 95	Com-plex FFT	H.264 Dec.
3×3	225	*3,202	529	40	24	305
4×4	393	7,951	1,248	184	41	571
5×5	561	†9,601	1,233	2,306	68	†1,417
6×6	161	†12,401	1,233	389	84	1,331
7×7	151	†22,477	1,233	2,018	†99	1,825
8×8	151	†12,417	1,233	†1,103	98	2,395
9×9	150	†22,476	1,232	1,192	†117	–
10×10	140	22,502	1,232	†1,466	†115	–
15×15	392	†16,960	1,232	†1,585	†111	–

4.6.2 Application-Specific Schedules

In this section we investigate the general case of scheduling arbitrary communication patterns. The communication patterns of interest are communication graphs from real applications. The MCSL NOC Benchmark Suite [9] provides statistical traffic patterns for seven different applications mapped onto different topologies of different sizes. The seven benchmarks represent different types of traffic patterns, such as one-to-many, many-to-many, grid-like patterns and combinations of these. These communication patterns are also valid for real-time systems, as they contain a dynamic control application, encoding and decoding of error-correcting codes and general mathematical operations.

The normalized communication patterns are scheduled with the presented scheduler on mesh and bi-torus topologies of different sizes. In Table 4.2 we show the obtained schedule period of the benchmark applications generated with ALNS and GRASP.

The long schedules shown in Table 4.2 are bound by IO of the most communication-intensive processors in the network. Another observation that can be made from the table is the irregularity of the schedule lengths. There is no correlation between the schedule period of an application mapped to a mesh topology or a bi-torus topology of the same size.

In practice it is not feasible to implement the hardware tables needed to support the longest schedules shown in Table 4.2 (1000+ time slots). Based on the hardware complexity of the NOC-implementation, we consider RAM or

ROM tables with 64-128 entries to be acceptable and tables with 256 entries to be a on the high side.

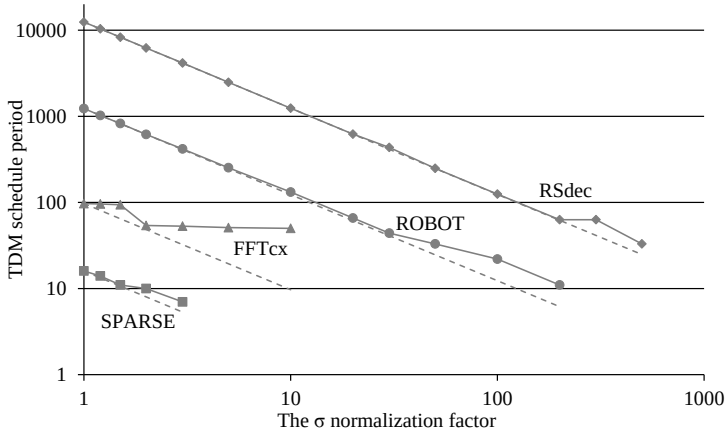


Figure 4.4: The measured and ideal schedule period as a function of σ of all the benchmark applications mapped on an 8×8 bi-torus platform.

Therefore, it is interesting how much the schedule period can be compressed before the overall performance decreases. Varying the normalization factor, we can observe how the resulting schedule periods change. In Figure 4.4 we show how the schedule period changes as a function of the normalization factor σ for the benchmark applications in an 8×8 bi-torus platform. We picked this topology because it reflects a likely topology. We have also experimented with a 16×16 bi-torus platform, where we found similar results. The dotted lines are the initial schedule period divided by the increasing σ. They indicate how the schedule period would decrease in the ideal case, where bandwidths are not affected (c.f., the discussion in Section 4.3). If the measurements are above the dotted line, the schedule period is longer than what corresponds to the compression factor. This represents a decrease in bandwidth (in the normalized domain) and this must be compensated for by an increase in the frequency of the TDM clock.

We see that the curves for all the applications follow the ideal lines well below 1000 time slots. When the curves break off from the ideal lines, this is because the increasing compression factor causes an over-allocation of bandwidth to the communication channels with the smallest requirements. It is seen that all applications can be scaled down to less than 100 TDM slots with a negligible performance degradation compared to the unscaled version. In most cases this can be compensated for by increasing the frequency of the TDM clock used in the NOC. We consider this insight and the idea of compressing the schedules an important contribution of the paper.

4.7 Discussion

Summarizing the results, we see that our scheduler produces very good results for the special case of all-to-all communication. The all-to-all schedules are hard to optimize because their link utilization is very high and their core communication graph is fully connected. The schedule period length of the all-to-all schedules grows polynimially with the number of nodes in the network. For large networks it is not be feasible to support an all-to-all schedule, because the latency is very high, the bandwidth to a single core is very limited, and the tables would be very large, increasing the size of the interconnect hardware. An application mapped onto a 16×16 platform, where all cores need to communicate to all other cores, is quite unlikely. In the tool flow, an all-to-all schedule would lead to simplifications. The mapping of tasks to processors is simplified because the bandwidth is equally low to all cores. Therefore, the mapping of tasks has very little effect on the performance, only the latency changes depending on the mapping.

From the application-specific schedules of the benchmark application, it looks plausible that we can set a limit on the number of time slots that an application-independent platform needs to support. For the applications in the benchmark suite it seems a good limit on the number of time slots is around 100. If we limit the number of time slots to 100, we can schedule all of the applications from the benchmark with only a negligible performance decrease compared to the unlimited case. Given a limit on the time slots available in the hardware platform, the tool flow should do a binary search for the optimal normalization factor.

Overall the application-specific schedules provide an excessive amount of bandwidth compared to the all-to-all schedules. The excessive amount of bandwidth can be removed by reducing the clock frequency of the network or reducing the width of the links, a combination of the two will save both power and area.

4.8 Conclusion

The paper presented a metaheuristic scheduler for inter-processor communication in multi-processor platforms using time division multiplexed (TDM) networks on chip (NOC). This scheduling problem is NP-complete and we use a metaheuristic approach to solve it. The scheduler is intended for use in a design flow where an application is mapped onto a predesigned and therefore fixed platform.

Input to the scheduler is a specification of the NOC in the platform and a specification of the application in the form of a core communication graph. The input formats are highly parameterized, and compared to previous work,

the scheduler therefore handles broader and more general class applications and platforms.

For the special case of all-to-all communication with identical bandwidth, our scheduler produces better results than reported in previous work. The scheduler was also evaluated for a set of larger and non-symmetric applications from the MCSL NOC benchmark suite. Among our results is the observation that our metaheuristics perform better than the greedy solution.

The scheduler uses a dimensionless and normalized representation of bandwidth requirements, and the paper shows that the period of the TDM schedule can be compressed to less than 100 slots with almost no increase in the TDM-clock frequency.

Acknowledgment

This work was partially funded by the Danish Council for Independent Research | Technology and Production Sciences, Project no. 12-127600: Hard Real-Time Embedded Multiprocessor Platform (RTEMP) project, and by the European Union's 7th Framework Programme under grant agreement no. 288008: Time-predictable Multi-Core Architecture for Embedded Systems (T-CREST).

Bibliography

[1] C. Blum and A. Roli. Metaheuristics in combinatorial optimization: Overview and conceptual comparison. *ACM Comput. Surv.*, 35(3):268–308, Sept. 2003. ISSN 0360-0300. doi: 10.1145/937503.937505.

[2] F. Brandner and M. Schoeberl. Static routing in symmetric real-time network-on-chips. In *Proceedings of the 20th International Conference on Real-Time and Network Systems*, RTNS '12, pages 61–70, New York, NY, USA, 2012. ACM. ISBN 978-1-4503-1409-1.

[3] E. Burke and G. Kendall. *Search methodologies: introductory tutorials in optimization and decision support techniques.* Springer Science+ Business Media, 2005.

[4] S. Even, A. Itai, and A. Shamir. On the complexity of time table and multi-commodity flow problems. In *Foundations of Computer Science, 1975., 16th Annual Symposium on*, pages 184 –193, oct. 1975.

[5] T. A. Feo and M. G. Resende. Greedy randomized adaptive search procedures. *Journal of Global Optimization*, 6(2):109–133, March 1995. ISSN 0925-5001. doi: 10.1007/BF01096763.

[6] S. Hansen. T-CREST project. Project webpage http://t-crest.org, 2012. URL `http://t-crest.org`.

[7] A. Hansson, K. Goossens, and A. Radulescu. A unified approach to mapping and routing on a network-on-chip for both best-effort and guaranteed service traffic. *VLSI Design*, 2007:16, 2007.

[8] E. Kasapaki, J. Sparsø, R. Sørensen, and K. Goossens. Router Designs for an Asynchronous Time-Division-Multiplexed Network-on-Chip. In *Proc. of Euromicro Conference on Digital System Design (DSD)*, pages 319–326, Sept. 2013.

[9] W. Liu, J. Xu, X. Wu, Y. Ye, X. Wang, W. Zhang, M. Nikdast, and Z. Wang. A noc traffic suite based on real applications. In *VLSI (ISVLSI), 2011 IEEE Computer Society Annual Symposium on*, pages 66 –71, july 2011.

[10] R. Marculescu, U. Ogras, L.-S. Peh, N. Jerger, and Y. Hoskote. Outstanding research problems in noc design: System, microarchitecture, and circuit perspectives. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 28(1):3–21, 2009. doi: 10.1109/TCAD.2008.2010691.

[11] M. Millberg, E. Nilsson, R. Thid, and A. Jantsch. Guaranteed bandwidth using looped containers in temporally disjoint networks within the nostrum network on chip. In *Design, Automation and Test in Europe Conference and Exhibition, 2004. Proceedings*, volume 2, pages 890 – 895 Vol.2, feb. 2004.

[12] S. Murali, G. D. Micheli, A. Jalabert, and L. Benini. XpipesCompiler: A tool for instantiating application specific networks-on-chip. In *Proc. Design, Automation and Test in Europe (DATE)*, pages 884–889. IEEE Computer Society Press, 2004.

[13] D. Pisinger and S. Ropke. Large neighborhood search. In M. Gendreau and J.-Y. Potvin, editors, *Handbook of Metaheuristics*, volume 146 of *International Series in Operations Research and Management Science*, pages 399–419. Springer US, 2010. ISBN 978-1-4419-1663-1.

[14] M. Schoeberl, P. Schleuniger, W. Puffitsch, F. Brandner, C. W. Probst, S. Karlsson, and T. Thorn. Towards a time-predictable dual-issue microprocessor: The Patmos approach. In *First Workshop on Bringing Theory to Practice: Predictability and Performance in Embedded Systems (PPES 2011)*, pages 11–20, Grenoble, France, March 2011.

[15] M. Schoeberl, F. Brandner, J. Sparsø, and E. Kasapaki. A statically scheduled time-division-multiplexed network-on-chip for real-time systems. In *Proceedings of the 6th International Symposium on Networks-on-Chip (NOCS)*, Lyngby, Denmark, May 2012. IEEE.

[16] J. Siek, L.-Q. Lee, and A. Lumsdaine. Boost graph library. http://www.boost.org/libs/graph/, June 2000.

[17] R. B. Sørensen, J. Sparsø, M. R. Pedersen, and J. Højgaard. A metaheuristic scheduler for time division multiplexed networks-on-chip. Technical Report DTU Compute Technical Report-2014-04, Technical University of Denmark, 2014.

[18] J. Sparsø, E. Kasapaki, and M. Schoeberl. An Area-efficient Network Interface for a TDM-based Network-on-Chip. In *Proc. Design Automation and Test in Europe (DATE)*, pages 1044–1047, 2013.

Chapter 5

Message Passing on a Time-predictable Multicore Processor

This chapter was previously published: © 2015 IEEE. Reprinted, with permission, from Rasmus Bo Sørensen, Wolfgang Puffitsch, Martin Schoeberl, and Jens Sparsø,
"Message Passing on a Time-predictable Multicore Processor", *IEEE 18th Symposium on Real-time Computing (ISORC)*, pages 51-59, 2015.

Abstract

Real-time systems need time-predictable computing platforms. For a multicore processor to be time-predictable, communication between processor cores needs to be time-predictable as well. This paper presents a time-predictable message-passing library for such a platform. We show how to build up abstraction layers from a simple, time-division multiplexed hardware push channel. We develop these time-predictable abstractions and implement them in software. To prove the time-predictability of these functions we analyze their worst-case execution time (WCET) with the aiT WCET analysis tool. We combine these WCET numbers with the calculation of the network latency of a message and then provide a statically computed end-to-end latency for this core-to-core message.

5.1 Introduction

In hard real-time systems, all tasks must meet their deadlines to avoid catastrophic consequences. Therefore, execution times of tasks, including communication timings, must have a provable upper bound. This provable upper bound is the worst-case execution time (WCET). The WCET is a major concern when the designer analyzes the performance of the system.

Inter-core communication via external shared memory quickly becomes a performance bottleneck in multicore processors as the number of cores grows. With a shared cache caching the external memory, this single resource still is a bottleneck. Access to a shared cache and cache-coherence traffic will not scale to more than a few processors. Using message passing between cores via a network-on-chip (NoC) promises to eliminate this bottleneck [14, 17]. Hardware support for on-chip message passing is also beneficial when added to a standard multicore architecture to reduce the cache traffic. Furthermore, the message passing NoC shall be visible and directly accessible to the application programmer for efficient and predictable use of the communication infrastructure.

The T-CREST project [25] developed a time-predictable multicore processor, consisting of the time-predictable processor Patmos [22], a time-predictable memory NoC [24, 3], a time-predictable memory controller [4, 15], and the time-predictable message passing NoC Argo [12, 29].

This paper mainly addresses the software layer for Argo. Argo uses time-division multiplexed (TDM) scheduling, which allows deriving upper bounds of message latencies [23, 27]. Furthermore, as the TDM schedule is static and precomputed [28], the routers and network interfaces are small. The network interfaces include direct memory access (DMA) controllers to transfer data from a local scratchpad memory (SPM) to a remote SPM. A processor can only setup a DMA to transfer data to a remote processor; this type of communication is called push communication. However, using the NoC for more general message passing between cores requires a detailed understanding of the hardware and its capabilities.

This paper presents a time-predictable message-passing (TPMP) library that abstracts from the details of the T-CREST platform and makes the platform's time-predictable features available to the application developer.

The Message Passing Interface (MPI) standard [18] – the de-facto standard for message passing in distributed memory systems – inspired the TPMP library. We used MPI as inspiration to provide an interface that is easy to use for developers that are already familiar with message passing. However, implementing MPI requires dynamic allocation of messages, which is usually avoided in real-time applications. For better analyzability we statically allocate message buffers. To avoid copying of data and maximize performance, our library operates on messages placed directly in the communication SPMs.

We implement flow control in software on top of the push communication supported in hardware. If we use the cycle executive model to implement hard real-time programs, one can argue that the library does not need flow control. However, the library needs flow control to implement atomic updates of sample values in state based communication. Flow control also simplifies communication between tasks that execute at different periods.

On top of the flow control we implement double buffering to interleave communication and computation. We need double buffering at both the sender and the receiver side to interleave communication and computation. By extending the double buffering to a queue of buffers we support multi-rate synchronous programming and asynchronous message passing.

Our library implements a barrier and a broadcast primitive in addition to the send and receive primitives. We envision that our platform developed for hard real-time applications will also support soft real-time and non-real-time application. Therefore, we explore a broader set of communication primitives including a barrier. Even though Argo does not provide direct hardware support for these primitives, the evaluation shows that our implementation is efficient.

The contributions of this paper are:

- a message passing library that takes into account the capabilities of the Patmos multicore processor while providing an interface that is familiar to developers
- an evaluation that shows efficiently implemented primitives with collective semantics on top of Argo, even though there is no direct hardware support for them
- an evaluation of the WCET of the implemented communication primitives

This paper is organized as follows: Section 5.2 presents related work. Section 5.3 presents background on the MPI standard. Section 5.4 presents an overview of the T-CREST hardware platform. Section 5.5 describes the design of the TPMP library. Section 5.6 describes the implementation of the TPMP library and provides evidence for its analyzability. Section 5.7 evaluates the WCET of TPMP library functions. Section 5.8 concludes the work presented in this paper.

5.2 Related Work

Intel created the Single-chip Cloud Computer (SCC) as a research chip to ease research on many-core architectures [8, 9]. Along the SCC, Intel provides a library for message passing via the NoC called RCCE. RCCE provides high-level functions and “gory” low-level functions for message passing [16].

Scheller [21] investigates real-time programming on the Intel SCC. In particular, he describes a message passing interface for the SCC and evaluates the achievable bandwidth. In contrast, this paper investigates a message passing library for a hard real-time platform with time-predictable hardware.

A more detailed evaluation of the Intel SCC reveals that its NoC can exhibit unbounded timing behavior under high contention [20]. Due to the TDM scheduling used in Argo, such behavior would be impossible on the platform considered in this paper.

Kang et al. [11] present an evaluation of an MPI implementation for the Tile64 processor platform from Tilera. The sending primitive loads the message data through the data cache, causing high cache miss costs for large messages. The library presented in our paper avoids these costs by placing messages in the communication SPM.

The CompSOC platform [5] aims at time-predictability, similarly to the platform we are targeting. While the hardware implementation of the NoC in CompSOC is more complex than in our platform, the network interfaces resemble each other from a software perspective: the application places messages in a local memory and transfers them through the NoC with a DMA mechanism. Therefore, the design of the library presented in this paper should also apply to the CompSOC platform.

There has been an attempt to define a variant of the MPI standard for real-time systems [10, 26]. However, this real-time variant of MPI has not found widespread adoption.

To analyze multicore programs using message passing, Potop-Butucaru et al. [19], describes a method that includes communication in the control flow graph of the program. The architecture and library we present in this paper can also use this method.

5.3 MPI background

The MPI standard [18] is the de-facto standard interface for message passing in distributed memory systems. The MPI standard has eight different communication concepts, where four of them apply for all versions of the MPI standard and the remaining four only apply to the MPI-2 version of the MPI standard. We have decided that the concepts of MPI-2 are out of scope for this paper, as we focus on the basic message passing primitives. The four communication concepts that apply for all the MPI standards are:

1. Communicator
2. Point-to-point basics
3. Collective basics

4. Derived datatypes

The Communicator concept describes a group of processors that can communicate. The program can reorganize a group during runtime. We omit the runtime configuration, as it is not statically analyzable. Instead we setup the communication channels statically to provide a statically analyzable solution.

The point-to-point concepts describe the send and receive functions in blocking and non-blocking versions. Point-to-point communication behaves as a communication channel through which only one core can send and only one core can receive. With TPMP we implement the principles of point-to-point communication of MPI. We provide both blocking and non-blocking versions of the send and the receive functions.

The concept of collective behavior describes how groups of processors can communicate. The collective communication involves both synchronization and exchange of data between multiple processors. The semantics of collective behavior in MPI is that they all start with a barrier to synchronize and then exchange data. By analyzing the sequential pieces of code between communication points and joining the results together in a system-level analysis, the designer can analyze the collective behavior. In this paper we present the implementation of the base services for collective behavior, the barrier and the broadcast. The other collective primitives are straightforward to implement using the principles from the barrier and the broadcast.

The derived datatypes concept defines some MPI specific datatypes that can have different implementations on different architectures. While the derived datatypes are useful on heterogeneous systems, they are out of scope for this paper because we focus on a homogeneous hardware platform.

The TPMP environment differs from the MPI standard in four ways: (1) TPMP only addresses on-chip communication. Therefore, the communication stack of our platform and TPMP can be much shallower than the communication stack of the MPI standard. (2) The T-CREST platform uses TDM to communicate through the NoC. Therefore, the latency and bandwidth of the communication channels can be guaranteed and are easy to compute. (3) The data structures should be statically allocated in the initialization phase before the application switches into hard real-time mode. (4) The shared memory bandwidth is the bottleneck of the system, so TPMP should force the programmer to keep data locally, reducing the use of the shared memory as much as possible.

5.4 The T-CREST Platform

This section gives an overview of the T-CREST multicore platform and a more in-depth presentation of the hardware functionality of the message passing NoC and the associated TDM scheduler.

5.4.1 Platform Overview

The Patmos multiprocessor is a time-predictable homogeneous multiprocessor platform. It is designed to be a general-purpose platform for real-time systems, but it is also possible to instantiate application-specific FPGA implementations.

Figure 5.1 shows the structure of a Patmos processor node. Each processor node contains three caches and three SPMs: a method cache (M$), a stack cache (S$) for stack allocated data, a data cache (D$) for heap allocated and static data, and SPMs for instructions, data, and message passing communication. Patmos can also bypass caches and directly access the shared memory. The platform has two NoCs, one that provides access to a shared memory, and one that supports inter-core communication. We refer to these NoCs as the *memory tree NoC* (due to its structure) and the *message passing NoC*. Both NoCs use TDM to guarantee latency and bandwidth.

The processor has two address spaces: (i) a globally shared address space, and (ii) a local I/O address space for I/O devices and local SPM data. Accesses to the globally shared address space go through the memory tree NoC.

5.4.2 The Message Passing NoC

The Argo packet switched NoC for message passing implements end-to-end virtual channels and DMA controllers in the processor nodes. The processor can set up a DMA controller to push a block of data from the local SPM into the SPM of a remote processor core. This is the fundamental hardware mechanism underlying our message-passing library. A range of multicore platforms including [13, 5] provide similar functionality to Argo.

The processor needs to set up a communication channel to communicate between two processors. The Argo NoC uses static TDM scheduling for routing communication channels in routers and on links. The repeating schedule is an assignment between communication channels, DMA controllers, and TDM slots. In every time slot the NI can transmit a short packet with a two-word payload. The NI sends larger blocks of data (i.e., messages) as a sequence of packets. In this way all the outgoing channels from a processor node can be active at the same time in a time-multiplexed fashion.

In contrast to other TDM based NoCs that require credit-based flow control across the virtual channels, we have been able to avoid all forms of flow control and buffering in hardware. We achieve this by a novel network interface (NI) design [29] that integrates the DMA controllers with the TDM scheduling in the NIs as illustrated in Figure 5.1. In a given time slot of the TDM schedule, the corresponding DMA controller reads two words of payload data from the SPM and injects a packet into the NoC. This packet traverses the NoC and when it arrives at the destination the NI writes it directly to the SPM.

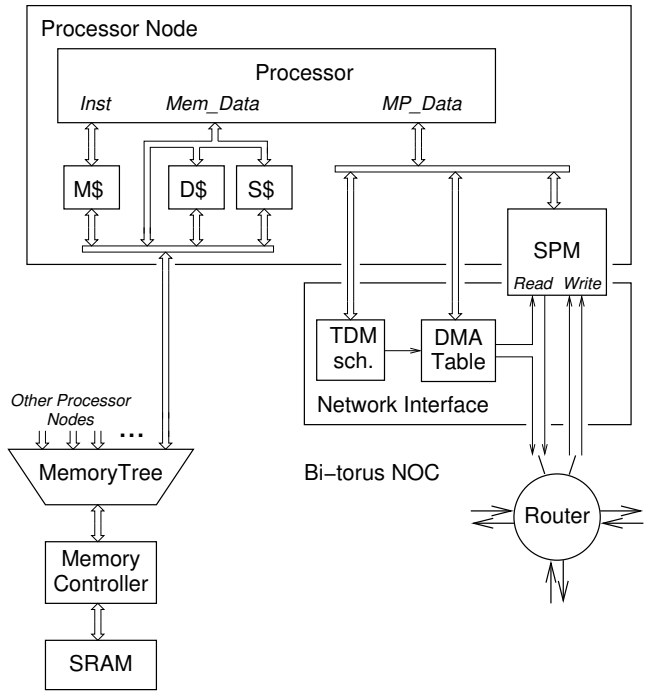


Figure 5.1: Block diagram of a node in the T-CREST multicore processor and its connections to the memory tree NoC and the message passing NoC.

We have implemented the DMA controllers in a single time-shared DMA state machine, because only one controller is active in each TDM slot. The design stores the address pointers and the word count corresponding to one logical DMA controller in a table. This sharing of the DMA controller hardware, and the absence of flow control and buffering, results in an extremely small hardware implementation; the NI is 2-3 times smaller than existing designs offering similar functionality [29]. At the same time the design supports a globally-asynchronous locally-synchronous platform with a minimum of clock-domain crossings.

5.4.3 The Scheduler

The off-line scheduler [28] generates a TDM schedule for a given application. Input to the scheduler is a communication graph that specifies groups of tasks mapped to the same processor (nodes in the graph), and the communication channels (edges annotated with the required bandwidth) along with a mapping of groups of tasks to specific processors in the platform.

Using a meta-heuristic optimization algorithm, the scheduler minimizes the period length of the generated schedule. In general, the schedule period depends on the number of processor nodes and channels. It is interesting to note that a schedule for a fully connected communication graph with the same bandwidth requirement on all channels is 19 slots for a 16-node platform and 85 slots for a 64-node platform.

The scheduler supports arbitrary NoC topologies, arbitrary pipelining in the individual routers and links, and different sized packets on different channels.

5.5 Design

A user-friendly message passing library shall hide all the complicated and the platform-specific details from the programmer, such that the programmer can concentrate on the application design. The programmer should not have to take care of hiding communication latency or preventing message buffers from overflowing. The library should hide these details from the programmer without inferring significant overhead and while maintaining the analyzability of the whole system. When we designed the TPMP library we assumed a platform similar to the T-CREST platform, with simple DMA-driven NIs [29].

5.5.1 Requirements

The overall requirements to the TPMP library are time-predictability, ease-of-use, and low overhead. To be time-predictable it shall be possible to compute the end-to-end latency of a message transfer. This end-to-end latency depends on the size of the message, the bandwidth allocated for the communication channel, and the code running on the processors involved in the communication. The communication primitives shall be implemented such that they minimize the WCET.

Ease-of-use means that the interface functionality shall be designed to fit many different applications. The interface shall provide communication primitives with different levels of configurability, such that the application developer can choose which runtime checks to perform in the application.

In a low overhead design it is important to avoid unnecessary movement of data. If the message data is moved to the shared memory it might be evicted from the caches, which can lead to a very high WCET. Even if the data stays resident in the cache, WCET analysis might not be able to classify those accesses as cache hits.

5.5.2 Push Communication

To push data to another core we need to setup a DMA transfer. We need four parameters to setup the DMA transfer: (1) the local address, (2) the destination core, (3) the remote address, and (4) the amount of data that the DMA should transfer. After this setup, the DMA and NoC transfer the message without any processor interaction. The sender can poll the DMA to detect the completion of the push message transfer.

On the receiving side the NI moves the message data to the destination address in the SPM without any interaction between the processor and the NI. The NoC and NI do not support any notification of a completed message transfer. Therefore, we need to implement this notification of the completed message transfer in software on top of the pure push communication. The NI transfers the message data in-order, so when the processor detects the last word of a transfer, it knows that it has received the complete message. Therefore, we append one word for a flag to the end of a message that is initially cleared by the receiver. The message itself has this flag set (by the sender). The receiver polls this flag to detect when the message has arrived. Then the flag is reset again.

5.5.3 Flow Control

To implement flow control, the receiver needs to acknowledge that it has received the previous message, such that the sender can send a new message. To avoid flow control on the acknowledge message, we need an acknowledgement scheme where consecutive acknowledgements can be overwritten without losing data. Such a scheme can be a simple counter, counting the number of messages the receiver has acknowledged. Every time the receiver acknowledges a message, the library updates the counter and sends the value of the counter to the sender. The sender can then calculate if there is any free buffer space at the receiver, by subtracting the number of acknowledged messages from the number of messages sent.

The acknowledgment message uses the very same push communication as described above for the message transfer. It is not different from a normal data packet.

5.5.4 Point-to-Point Primitives

Point-to-point communication involves two processors, the sender and the receiver. To make point-to-point communication efficient, the library needs to double buffer the messages to interleave communication and computation. To take advantage of the overlapping in both ends of the point-to-point communication channel, both the sender and receiver need to have two buffers. These

four buffers comprise one for the sender to write into, two for the network to operate on, and one for the receiver to read from. The sender needs to keep track of its own double buffer and it needs to coordinate with the receiver where to write into the receiver's double buffer.

We construct the double buffering as a circular buffer. To control the circular buffer at the receiver the sender needs a pointer to the tail of the queue and the receiver needs a pointer to the head of the queue. With the flow control described in Subsection 5.5.3, the only check the primitives have to make is to reset the head or tail pointer when they reach the end of the circular buffer space.

To increase the flexibility of the point-to-point communication primitives and to support a larger set of programming models, we add buffers to the existing double buffer and use the available handling of the circular buffers. The number of buffers is configurable on a per channel basis.

We can add additional capacity either to the sender side or to the receiver side. If we add the buffer capacity to the sender circular buffer, the sending processor needs to handle every message twice. Once to enqueue the message and once to setup the DMA transfer for that message. Therefore, we add the buffer capacity to the receiver circular buffer where we can wait for a message and dequeue the message in a single step. As we already have circular buffers for the double buffering, it is straightforward to increase the size of the circular buffers.

The acknowledgement scheme described in Subsection 5.5.3 together with the added buffer capacity allows the receiver to receive two or more messages before acknowledging any of these messages.

Figure 5.2 shows the memory layout of the two communication data structures allocated in the SPMs. The application programmer can configure the number of read buffers of the point-to-point channel to match the needs of the application. The library hides the communication latency from the sender, by overlapping computation and communication in two write buffers. The *Acknowledge count* in the receiver SPM is the number of messages that the receiver has acknowledged. The acknowledge primitive transfers the local *Acknowledge count* to the remote *Acknowledge count* in the sender SPM using push communication. The sender can compute the number of free buffers in the receiver side message queue from the number of messages sent and the number of messages acknowledged.

After using the data in a buffer, the receiver sends the acknowledgement for this buffer back to the sender. An acknowledgement means that the point-to-point connection can reuse the acknowledged receive buffer for new data.

For the point-to-point communication there are three primitives, (1) a send primitive, (2) a receive primitive, and (3) an acknowledge primitive. For each primitive there is a blocking and a non-blocking version. The blocking commu-

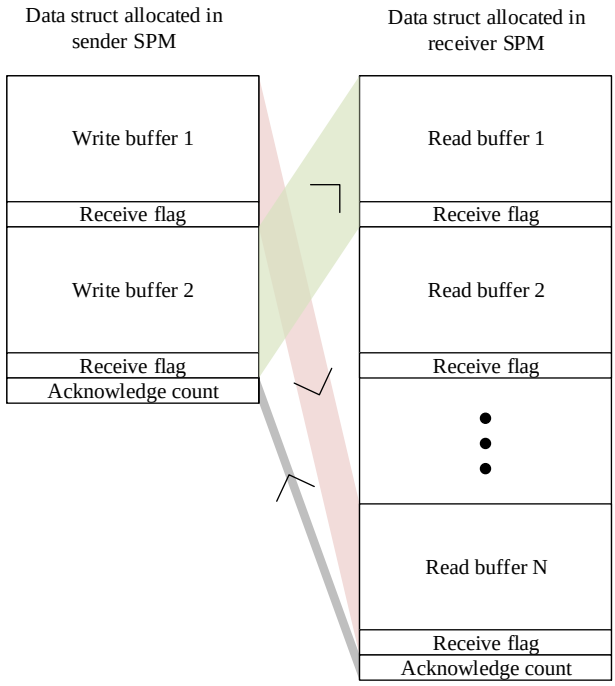


Figure 5.2: The memory layout of the buffering structure for point-to-point communication primitives.

nication primitives complete when all conditions are meet. The non-blocking versions check the conditions and return a success or a failure code depending on the failing condition.

The non-blocking send primitive can fail for two reasons, either there is no free buffer at the receiver end, or there is no free DMA on the sender side to transfer the data. The non-blocking receive and acknowledge primitives can each fail for one reason. If the buffer queue is empty, the non-blocking receive primitive fails. If there is no free DMA to transfer the *Acknowledge count* to the sender, the non-blocking acknowledge primitive fails. With the error codes the application programmer can take action depending on the error.

5.5.5 Collective Primitives

The basic collective primitive is a barrier. All cores in a group call the barrier function for synchronization. The semantics of a barrier is that none of the participating tasks can advance beyond the barrier before all tasks have called the barrier function.

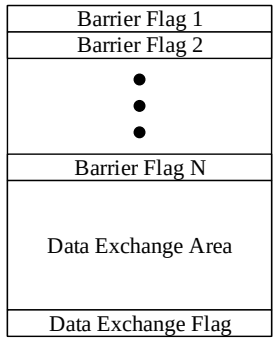


Figure 5.3: The memory layout of the collective communication primitives.

Broadcast, all-to-all, reduce, and reduce-all are examples of extended primitives. The semantics of these extended primitives are a barrier synchronization followed by a data exchange. The differences between these extended primitives are the data exchange patterns. We have implemented the broadcast to show the basic operation, and with the following ideas we can easily implement the other extended collective primitives. As the extended collective primitives start with a barrier, the tasks have to finish using the data of the previous exchange before calling the next collective primitive. Therefore, there is no need for flow control; we only need a packet arrival notification. Or in other words, the barrier serves as flow control.

Barrier A barrier has two phases: notify and wait. In the notify phase the task will notify the task group that it has reached the barrier. In the wait phase the task will wait for a notification from all the tasks in the group. When a task has seen all members of its group arrive at the barrier, it can continue its execution. We can implement a notification to all members of a group by sending a message to each of them. With an all-to-all TDM schedule in the NoC the bandwidth is already allocated and the individual communication channels do not interfere. Therefore, each member in the group sends a flag to every other member. Figure 5.3 shows the memory layout for the collective primitives in each SPM. The barrier uses only the barrier flags.

Broadcast With the help of the barrier we can implement a broadcast. The broadcast starts with a barrier and then the root process of the broadcast transmits a block of data to all the other participants in the broadcast. We place the broadcast data in the data exchange area of the root process, as seen in Figure 5.3. The broadcast primitive transfers this data to the participants' SPMs by setting up a DMA transfer for each participant.

5.6 Analyzable Implementation

Our main target for the implementation of the TPMP library is its WCET analyzability. During the development we use AbsInt's aiT WCET analysis tool [6] to guide development decisions. First, we bound all loops to enable WCET analysis. Second, cache misses are very costly on a time-predictable multicore processor. Therefore, we tried to avoid accessing shared memory completely, by allocating as many data structures as possible in the processor local SPM.

5.6.1 Push Communication

The Argo NoC implements push communication in hardware, but does not generate a notification when a message is received. We implement the receive notification in a single 32-bit value at the end of each message. Adding the receive notification does not change the way we analyze the communication.

5.6.2 Flow Control

Our design implements flow control by sending a counter value from the receiver to the sender. The library sends this counter value in a single network flit (64 bits of unsigned data), with no receive flag as notification. Calculating the number of free buffers is safe across an overflow, as long as the overflow value is greater than the largest possible difference between the two unsigned values, i.e., the number of buffers in the queue.

5.6.3 Point-to-Point Primitives

Figure 5.4 shows the interaction between two communicating threads using the blocking point-to-point primitives. When `mp_send()` sets up the DMA, the NI starts to transmit packets to the receiver. After receiving all packets the blocking `mp_recv()` continues. When the receiver finishes using the received message, it updates the *Acknowledge count* and sends it to the sender. Depending on the number of free elements in the buffer queue, the sender may proceed.

To analyze the blocking primitives, we assume that the blocking primitives do not have to wait for messages to arrive or free buffer space. We bounded the unbounded `while` loops in the blocking point-to-point primitives with source code annotations. We set the upper loop bound of the `while` loops to one. The analysis method presented in [19] supports this interaction enabling worst-case response time analysis.

The implementation of the non-blocking point-to-point primitives is free of unbounded loops. Therefore, the source code needs no annotations to complete

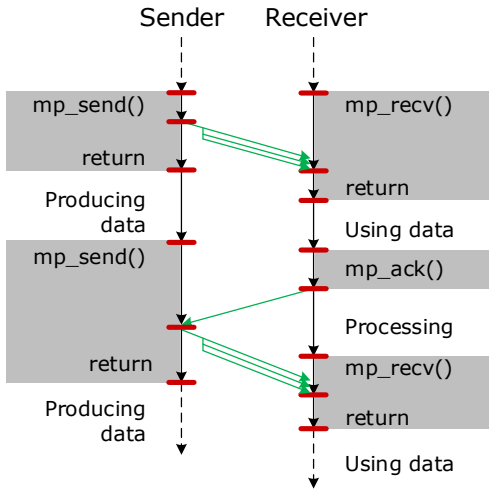


Figure 5.4: A model of the implemented point-to-point communication.

the analysis. We minimized the WCET for all primitives by looking at the feedback from the interactive analysis in the aiT tool.

5.6.4 Barrier Primitive

Figure 5.5 shows the interaction between cores that participate in a barrier. First, the barrier preamble calculates the addresses of the flags to send to the other participants. Then, it sends a message with a flag to all the others. When the primitive has set up all messages for transfer, the core synchronizes with the other cores one by one. To separate subsequent barrier calls, the primitive needs to reset the flag; resetting the flag requires the cores to synchronize a second time. To avoid resetting the flag twice, we make use of sense switching, first described by Hensgen [7]. Sense switching combines an alternating phase with the flag.

5.6.5 Broadcast Primitive

Figure 5.6 shows the model of the broadcast primitive. The broadcast primitive starts by synchronizing all cores with a barrier call. After the barrier call, the root of the broadcast pushes data to the other cores by setting up one DMA transfer to each of them. The cores receiving data from the root core wait for the receive flag.

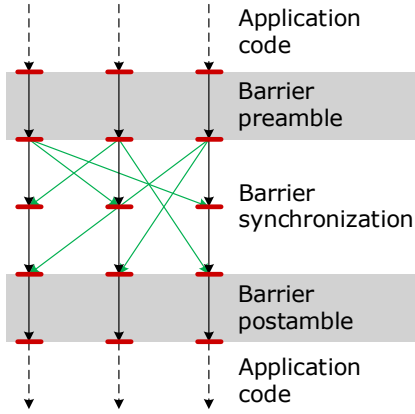


Figure 5.5: A model of the implemented barrier primitive.

5.6.6 Concurrency Issues

The SPM has two independent read/write ports that are used by the processor and by the NI. The processor and the NI behave as two truly concurrent threads: a software thread and a hardware thread. Both threads can access the same memory cell in the exact same clock cycle. Reading concurrently is not an issue. Writing concurrently will result in an undefined value. The TPMP library avoids concurrent writes by design. The remaining issue is reading and writing concurrently.

Most FPGA technologies do not define the result of a read during a write to the same address at the same time. An undefined value may be a random mix of the old value and the new value, but the stored value will be the new value [1, 30]. Reading an undefined value from the SPM might cause wrong behavior by the communication primitives.

In the design, discussed in Section 5.5, a processor can read an undefined value when reading the receive flag or the acknowledge count. The DMA controller of the NI will never read an undefined value as the processor starts the DMA only after the processor has written all the data.

If the NI and the processor reside in the same clock domain, we can solve the problem of reading an undefined value in hardware by adding forwarding to the SPM, from the network port to the processor port; but with the implementation of our TPMP library this is not necessary.

The receive flag arrives only after the NoC has delivered all the message data. If the processor reads the receive flag in the same cycle as the flag arrives, the processor reads an undefined value; 0 or 1. If the processor reads a 1 it correctly concludes that it has received a message. If it reads a 0 it will continue

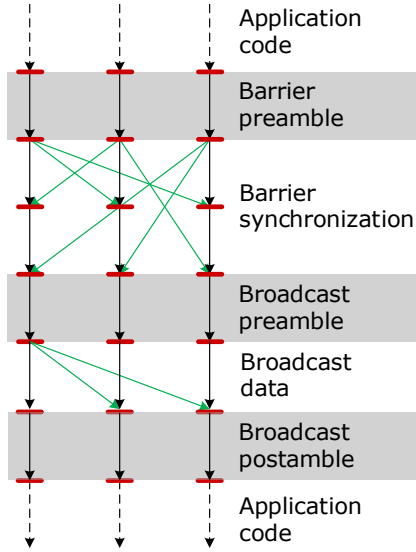


Figure 5.6: A model of the implemented broadcast primitive.

waiting and polling at the next try it will read a 1. This adds only a few cycles to the receive operation.

For the acknowledgment the receiver communicates back to the sender, the situation is somewhat similar, but more complex. The count value may signal the availability of one or more free buffers and the processor may read an arbitrary value.

A key observation is that this only happens if the NI writes a new value in the same cycle, and that this only happens when there is at least one free buffer in the receiver. By restricting to sending only a single message after a read of the count value, the sending processor can draw one of two conclusions and both are on the safe side: (i) If a potentially incorrect count value causes the sender to conclude that the receiver does not have a free buffer, then the sender will continue waiting and polling, and at the next read of the count it will read the correct value. (ii) If the potentially incorrect count value causes the sender to conclude that the receiver does have at least one free buffer, then – despite the incorrect count value – this conclusion is actually correct. In both cases the behavior is correct, and in the worst case the additional polling operation adds a few cycles to the latency of the send operation.

If the processor and NI are in different clock domains it is not only a matter of reading potentially undefined digital values. It is also a matter of metastability and reading non-digital signals. Handling of this situation requires synchronization of signals/flags to the processor clock domains. Implementing

this involves a minor addition to the NI implementation and this is future work. However, the effect on the WCET analysis as presented in this paper will be very small.

5.7 Evaluation

For the WCET analysis and the measurements we use a 9-core platform with a 3×3 bi-torus network and a TDM arbiter for shared memory access. Each core in the platform is running at 80 MHz and has 4 KB of communication SPM. The platform is running in an Altera Cyclone IV FPGA with 2 MB of external SRAM. The cache miss time with the TDM arbiter is 189 clock cycles for a 16-byte burst.

We computed the WCET numbers with the aiT tool from AbsInt [6], which supports the Patmos processor architecture. For the average-case execution time (ACET) results, we ran a test application in the described hardware, reading out the clock cycle counter to get the timing. For these results we assume that the functions are resident in the method cache.

We optimized the source code of the TPMP library functions with respect to WCET by looking at the feedback from the interactive mode of the aiT tool.

5.7.1 Point-to-Point Primitives

Table 5.1 shows the measured ACET and the WCET of all library functions. Each blocking function contains a `while` loop that blocks until a condition becomes true. The design section describes the conditions of each primitive. We assume that the functions do not wait for any of these conditions to become true. A system-level analysis will show if our assumption does not hold and in this case we can add the delay found by the system level analysis to the WCET. For the WCET analysis we bounded the loop iteration count to one. As shown in Table 5.1 the WCETs of the blocking function calls are higher than the WCETs for the non-blocking calls. This is because the *if* statement in the non-blocking primitive use predicates, avoiding a conditional branch.

The WCET is relatively close to the measured average-case execution time because the platform is optimized for time predictability. The execution time is a few tens of clock cycles. The library code is efficient, as no data needs to be copied between the user program and the message-passing library. All buffers in the SPM are directly usable for computation and communication.

5.7.2 Barrier Comparison

Figure 5.7 shows the WCET and the ACET of our message-passing barrier against a tournament barrier [7]. The hardware platform is the same for all

Table 5.1: Average-case execution time (ACET) and worst-case execution time (WCET) for each point-to-point communication primitive.

Function	ACET (cycles)	WCET (cycles)
`mp_nbsend()`	51	83
`mp_send()`	74	99
`mp_nbrecv()`	32	36
`mp_recv()`	28	43
`mp_nback()`	55	59
`mp_ack()`	49	77

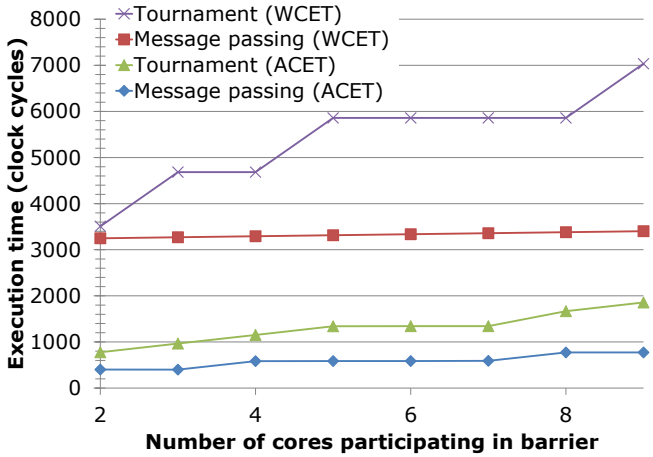


Figure 5.7: The measured average-case execution time and the worst-case execution time of a barrier call as a function of the number of cores participating in the barrier.

the measurements; the number of cores is the number of cores participating in the barrier. This figure shows that our barrier implementation using message passing is faster than the shared memory tournament barrier in both the worst case and the average case. Furthermore, message passing scales better in the number of cores.

5.7.3 End-to-End Latency

We can calculate the end-to-end latency of transmitting a message from one core to another with our library by adding the WCET of the executed code and the time it takes the DMA to transfer the data. We refer to this latency as L_{msg}. Gangwal et al. [2] show how to calculate the latency of a write transaction

through a TDM NoC. With an all-to-all TDM schedule, where all communication channels have equal bandwidth, this formula simplifies to what we show below.

We assume that the clock frequency of the processors and the NoC is the same to shorten the formula. One could extend the formula to account for multiple frequencies, but we omit this here for space reasons. L_{msg} is the worst-case latency in clock cycles from the time the source processor calls `mp_send()` to the time the destination processor returns from `mp_recv()`, assuming that the sender does not wait for a free buffer or a free DMA, and the receiver is ready to call `mp_recv()`. A system-level analysis can find any delays that break these assumptions and add them to the worst-case latency.

L_{msg} consists of two parts: (1) the WCET of the code running on the processors and (2) the latency of a write transaction [2]. Table 5.1 shows the WCET of the communication primitives and Equation 5.1 shows the formula for the latency L_{write} of a write transaction.

$$L_{\text{write}} = \left(\left\lceil \frac{S_{\text{msg}}}{S_{\text{chan}}} \right\rceil \cdot P_{\text{TDM}} \right) \cdot C_{\text{slot}} + H \cdot D \tag{5.1}$$

S_{msg} is the size of the transmitted message, S_{chan} is the number of payload bytes that the NoC can send in one TDM period from the source processor to the destination processor, P_{TDM} is the length of the TDM period, C_{slot} is the number of clock cycles in a TDM slot, H is the number of hops from the source to the destination processor, and D is the number of phits that one router can store.

With our Argo NoC S_{chan} is 8 bytes and C_{slot} is 3 clock cycles, meaning that two 32-bit words can transferred every 3 clock cycles. For the synchronous version of the Argo router D is 3 phits. For the presented 3×3 core platform H is at most 3 hops. With an all-to-all schedule for this platform, P_{TDM} is 10 time slots. S_{msg} is the message size.

To the latency of a DMA transfer we add the WCET of sending and receiving the message. The WCET of sending and receiving does not depend on the size of the message, because it does not involve moving the data. Table 5.2 shows the worst-case latency of sending a message from a sender to a receiver. The designer can reduce the latency of transmitting large messages considerably by generating an application specific schedule that reduces P_{TDM}.

5.8 Conclusion

Real-time systems need time-predictable computing platforms. For a multicore processor not only the processor needs to be time-predictable, but also the message passing hardware and software. This paper presented a message-passing

Table 5.2: The worst-case latency in clock cycles of sending a message, without acknowledgement of messages, for the blocking and non-blocking communication primitives.

Message size (bytes)	8	32	128	512	2048
Blocking	211	301	661	2101	7861
Non-blocking	188	278	678	2078	7838

library for a time-division multiplexed network-on-chip. We developed the library to be time-predictable and we show this by analyzing the code with the aiT WCET analysis tool from AbsInt. As the design carefully avoids access to shared memory in the library code, the resulting WCET for the message passing primitives is in the order of tens of clock cycles.

The message passing library and the application code operate on data allocated in a local scratchpad memory that the network-on-chip also use for data transmission. Therefore, the message passing library does not need to copy data and the WCET of the message passing functions is less than 100 clock cycles, independent of the message size.

Acknowledgments

The work presented in this paper was funded by the Danish Council for Independent Research | Technology and Production Sciences under the project RTEMP,[1] contract no. 12-127600.

Source Access

The presented work is open source and the full T-CREST tool chain can be downloaded from GitHub and built under Ubuntu with the following two commands:

```
git clone https://github.com/t-crest/patmos-misc.git misc
./misc/build.sh
```

[1] http://rtemp.compute.dtu.dk

Bibliography

[1] A. Corporation. Stratix V device handbook, volume 1: Device interfaces and integration. Technical report, Altera Corporation, 2014. URL http://www.altera.com/literature/hb/stratix-v/stratix5_handbook.pdf.

[2] O. Gangwal, A. Rădulescu, K. Goossens, S. González Pestana, and E. Rijpkema. Building predictable systems on chip: An analysis of guaranteed communication in the aethereal network on chip. In P. van der Stok, editor, *Dynamic and Robust Streaming in and between Connected Consumer-Electronic Devices*, volume 3 of *Philips Research*, pages 1–36. Springer Netherlands, 2005. ISBN 978-1-4020-3453-4. doi: 10.1007/1-4020-3454-7_1.

[3] J. Garside and N. C. Audsley. Investigating shared memory tree prefetching within multimedia noc architectures. In *Memory Architecture and Organisation Workshop*, 2013.

[4] M. D. Gomony, B. Akesson, and K. Goossens. Architecture and optimal configuration of a real-time multi-channel memory controller. In *Design, Automation Test in Europe Conference Exhibition (DATE), 2013*, pages 1307–1312, 2013. doi: 10.7873/DATE.2013.270.

[5] K. Goossens, A. Azevedo, K. Chandrasekar, M. D. Gomony, S. Goossens, M. Koedam, Y. Li, D. Mirzoyan, A. Molnos, A. B. Nejad, A. Nelson, and S. Sinha. Virtual execution platforms for mixed-time-criticality systems: The CompSOC architecture and design flow. *SIGBED Rev.*, 10(3):23–34, Oct. 2013. ISSN 1551-3688. doi: 10.1145/2544350.2544353. URL http://doi.acm.org/10.1145/2544350.2544353.

[6] R. Heckmann and C. Ferdinand. Worst-case execution time prediction by static program analysis. Technical report, AbsInt Angewandte Informatik GmbH. URL http://www.absint.de/aiT_WCET.pdf. [Online, last accessed November 2013].

[7] D. Hensgen, R. Finkel, and U. Manber. Two algorithms for barrier synchronization. *International Journal of Parallel Programming*, 17(1):1–17, 1988. ISSN 0885-7458. doi: 10.1007/BF01379320. URL http://dx.doi.org/10.1007/BF01379320.

[8] Intel Labs. SCC external architecture specification (EAS). Technical report, Intel Corporation, May 2010.

[9] Intel Labs. The SCC programmer's guide. Technical report, Intel Corporation, January 2012.

[10] A. Kanevsky, A. Skjellum, and A. Rounbehler. Mpi/rt-an emerging standard for high-performance real-time systems. In *System Sciences, 1998., Proceedings of the Thirty-First Hawaii International Conference on*, volume 3, pages 157–166. IEEE, 1998.

[11] M. Kang, E. Park, M. Cho, J. Suh, D. Kang, and S. P. Crago. MPI performance analysis and optimization on Tile64/Maestro. In *Proceedings of Workshop on Multi-core Processors for Space—Opportunities and Challenges Held in conjunction with SMC-IT*, pages 19–23, 2009.

[12] E. Kasapaki, J. Sparsø, R. B. Sørensen, and K. Goossens. Router designs for an asynchronous time-division-multiplexed network-on-chip. In *Digital System Design (DSD), 2013 Euromicro Conference on*, pages 319–326. IEEE, 2013.

[13] M. Kistler, M. Perrone, and F. Petrini. Cell multiprocessor communication network: Built for speed. *Micro, IEEE*, 26:10–25, 2006. URL `http://ieeexplore.ieee.org/iel5/40/34602/01650177.pdf`.

[14] R. Kumar, T. G. Mattson, G. Pokam, and R. Van Der Wijngaart. The case for message passing on many-core chips. In M. Hübner and J. Becker, editors, *Multiprocessor System-on-chip: Hardware Design and Tool Integration*, chapter 5, pages 115–123. Springer, 2011.

[15] E. Lakis and M. Schoeberl. An SDRAM controller for real-time systems. In *Proceedings of the 9th Workshop on Software Technologies for Embedded and Ubiquitous Systems*, 2013. URL `http://www.jopdesign.com/doc/sdramctrl.pdf`.

[16] T. Mattson and R. van der Wijngaart. RCCE: a small library for many-core communication. Technical report, Intel Corporation, 2010.

[17] T. G. Mattson, R. F. Van der Wijngaart, M. Riepen, T. Lehnig, P. Brett, W. Haas, P. Kennedy, J. Howard, S. Vangal, N. Borkar, G. Ruhl, and S. Dighe. The 48-core SCC processor: the programmer's view. In *International Conference for High Performance Computing, Networking, Storage and Analysis (SC)*, pages 1–11, 2010. ISBN 9781424475575.

[18] MPI-forum. *MPI: A Message-Passing Interface Standard Version 3.0*. MPI-forum, 2012. URL `http://www.mpi-forum.org/docs/mpi-3.0/mpi30-report.pdf`. Available at `http://www.mpi-forum.org/docs/mpi-3.0/mpi30-report.pdf`.

[19] D. Potop-Butucaru and I. Puaut. Integrated Worst-Case Execution Time Estimation of Multicore Applications. In C. Maiza, editor, *13th International Workshop on Worst-Case Execution Time Analysis*, volume 30 of

OpenAccess Series in Informatics (OASIcs), pages 21–31, Dagstuhl, Germany, 2013. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik. ISBN 978-3-939897-54-5. doi: http://dx.doi.org/10.4230/OASIcs.WCET.2013.21. URL http://drops.dagstuhl.de/opus/volltexte/2013/4119.

[20] W. Puffitsch, E. Noulard, and C. Pagetti. Mapping a multi-rate synchronous language to a many-core processor. In *Proceedings of the 19th Real-Time and Embedded Technology and Applications Symposium*, RTAS '13, pages 293–302. IEEE, 2013. doi: 10.1109/RTAS.2013.6531101.

[21] J. Scheller. Real-time operating systems for many-core platforms. Master's thesis, Institut supérieur de l'aéronautique et de l'espace, Toulouse, France, 2012.

[22] M. Schoeberl, P. Schleuniger, W. Puffitsch, F. Brandner, C. W. Probst, S. Karlsson, and T. Thorn. Towards a Time-predictable Dual-Issue Microprocessor: The Patmos Approach. In *Bringing Theory to Practice: Predictability and Performance in Embedded Systems*, volume 18 of *OASICS 18 Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik*, pages 11–21, Mar 2011. URL http://hal.inria.fr/inria-00585320/PDF/schoeberl-ppes11.pdf.

[23] M. Schoeberl, F. Brandner, J. Sparsø, and E. Kasapaki. A statically scheduled time-division-multiplexed network-on-chip for real-time systems. In *Proceedings of the 6th International Symposium on Networks-on-Chip (NOCS)*, pages 152–160, Lyngby, Denmark, May 2012. IEEE. doi: 10.1109/NOCS.2012.25. URL http://www.jopdesign.com/doc/s4noc.pdf.

[24] M. Schoeberl, D. V. Chong, W. Puffitsch, and J. Sparsø. A time-predictable memory network-on-chip. In *Proceedings of the 14th International Workshop on Worst-Case Execution Time Analysis (WCET 2014)*, 2014.

[25] M. Schoeberl, C. Silva, and A. Rocha. T-CREST: A time-predictable multi-core platform for aerospace applications. In *Proceedings of Data Systems In Aerospace (DASIA 2014)*, Warsaw, Poland, June 2014.

[26] A. Skjellum, A. Kanevsky, Y. S. Dandass, J. Watts, S. Paavola, D. Cottel, G. Henley, L. S. Hebert, Z. Cui, and A. Rounbehler. The real-time message passing interface standard (MPI/RT-1.1). *Concurrency and Computation: Practice and Experience*, 16(S1):Si–S322, 2004. ISSN 1532-0634. doi: 10.1002/cpe.744. URL http://dx.doi.org/10.1002/cpe.744.

[27] R. B. Sørensen, M. Schoeberl, and J. Sparsø. A light-weight statically scheduled network-on-chip. In *Proceedings of the 29th Norchip Conference*, Copenhagen, November 2012. IEEE. URL http://www.jopdesign.com/doc/s4noceval.pdf.

[28] R. B. Sørensen, J. Sparsø, M. R. Pedersen, and J. Hojgaard. A metaheuristic scheduler for time division multiplexed networks-on-chip. In *Object/Component/Service-Oriented Real-Time Distributed Computing (ISORC), 2014 IEEE 17th International Symposium on*, pages 309–316, June 2014. doi: 10.1109/ISORC.2014.43.

[29] J. Sparsø, E. Kasapaki, and M. Schoeberl. An area-efficient network interface for a TDM-based network-on-chip. In *Design, Automation Test in Europe Conference Exhibition (DATE), 2013*, pages 1044–1047, March 2013. doi: 10.7873/DATE.2013.217.

[30] I. Xilinx. Spartan-6 FPGA block RAM resources user guide. Technical report, Xilinx, Inc., 2011. URL `http://www.xilinx.com/support/documentation/user_guides/ug383.pdf`.

Chapter 6

State-based Communication on Time-predictable Multicore Processors

This chapter is submitted to RTNS 2016 as:
Rasmus Bo Sørensen, Martin Schoeberl, and Jens Sparsø,
"State-based Communication on Time-predictable Multicore Processors".

Abstract

Some real-time systems use a form of task-to-task communication called state-based or sample-based communication that does not impose any flow control among the communicating tasks. The concept is similar to a shared variable, where a reader may read the same value multiple times or may not read a given value at all. This paper explores time-predictable implementations of state-based communication in network-on-chip based multicore platforms through five algorithms. With the presented analysis of the implemented algorithms, the communicating tasks of one core can be scheduled independently of tasks on other cores. Assuming a specific time-predictable multicore processor, we evaluate how the read and write primitives of the five algorithms contribute to the worst-case execution time of the communicating tasks. Each of the five algorithms has specific capabilities that make them suitable for different scenarios.

6.1 Introduction

In real-time systems, a variation of channel-based communication that emphasizes modularity and encapsulation is often denoted by phrases and terms like *sending of state messages* [10] or *sample based* communication [1]. The semantics of these types of *state-based* communication resembles a shared variable that is accessed atomically by a single writer and one or more readers without any coordination. A value may be read multiple times or not at all before it is overwritten by the next value.

In multicore systems, the number of processor cores is increasing year by year. Therefore, the access to a shared memory not only becomes a bottleneck, it also becomes more difficult to estimate the worst-case access time due to interference with memory traffic from other processor cores. In hard real-time systems, where the ability to find a tight estimate of the worst-case execution time (WCET) is of paramount importance, communication through the main memory represents a major problem. In this paper we concentrate on on-chip communication through a network-on-chip (NoC).

The contributions of this paper are: (i) a study of five WCET-analyzable state-based communication algorithms that target and exploit NoC-based multicore platforms and aim at minimizing interference, (ii) an analysis of the worst-case communication delay (WCCD) that is valid for any schedule where tasks meet their deadline, allowing the set of tasks to be scheduled independently of the task schedules on other cores, and (iii) an analysis of the worst-case end-to-end latency, presented through an example.

The schedulability analysis scales better to many cores if the tasks of one core can be scheduled independently of the tasks executing on other cores. Our algorithms offer modularity through independent timing and schedulability analysis of the communicating tasks on individual cores.

The paper is organized as follows: Section 6.2 provides related work on state-based communication and timing analysis. Section 6.3 describes our system model and hardware platform. Section 6.4 describes the five communication algorithms. Section 6.5 presents the analysis of the WCCD through a communication flow. Section 6.6 presents the analysis of the maximum end-to-end latency of an application using an example. Section 6.7 evaluates the presented algorithms. Section 6.8 concludes the paper.

6.2 Related Work

The concept of state-based communication is equivalent to a shared variable that can be written by a single writer process and read by multiple reader processes. Lamport [14] named a more general version of this problem, allowing multiple writer processes, the *readers/writers problem.*

Courtois et al. [3] first formulated the readers/writers problem and proposed two algorithms that prioritize either the reader processes or the writer processes. Both algorithms can make the processes that are not prioritized wait indefinitely, which is not acceptable in a real-time system.

Sorenson et al. [24] later proposed a non-blocking algorithm of the readers/writers problem for real-time systems, but their algorithm is limited to simulated concurrency using a single processor and it is not applicable on a multicore platform offering true concurrency.

Lamport [14] later proposed an algorithm allowing readers to read concurrently while a writer is writing. The algorithm allows the reader to check if a value is a mix of an old and a new value. In case of a mix, the reader re-reads. Lamport's algorithm does not use mutual exclusion, but continuous re-reading of the variable can keep the reader busy indefinitely. Kopetz and Reisinger [11] present an implementation of this algorithm, including an analysis of the upper bound on the number of re-reads due to writer interference. In the worst-case reader needs to read the whole message a number of times depending on the timing of the reader and writer, for large messages this overhead becomes prohibitively high and introduces a large amount of jitter.

In general-purpose systems, the predominant way of achieving mutual exclusion is to use a lock. As a solution to the readers/writers problem, Mellor-Crummey and Scott [17] proposed a read-write lock that allows multiple readers to gain access to the lock at the same time and thereby read simultaneously. Krieger et al. [12] proposed a similar version of the read-write lock that requires fewer atomic operations. For these locks to be usable in a real-time system, Brandenburg and Anderson [2] presented bounds of the blocking time of these read-write locks.

The timing analysis of communicating tasks in a real-time application can be divided into two parts: the WCET analysis of the application tasks and the analysis of the delay of communication between tasks. In a multicore system, the analysis of communication delays through a network is specific to the hardware implementation. Indrusiak [8] shows how response-time analysis can be used to calculate end-to-end latency of communication through a NoC with priority-preemptive arbitration in routers.

Gangwal et al. [7] shows how to calculate the communication delay of read and write transactions in a network using a time-division multiplexing (TDM) schedule.

With the WCETs and communication delays of the application, the end-to-end timing properties can be verified. Lauer et al. [15] presents such an end-to-end latency and freshness analysis of an integrated modular avionics systems. We show an example of a similar approach with a more detailed task model.

6.3 System Model

This section describes the semantics of state-based communication, the platform model and evaluation platform, and the application model that we assume in our analysis of the problem and in the analysis of our proposed algorithms.

6.3.1 State-based Communication

The concept of state-based communication is vaguely defined in the literature. The fundamental mechanism involves a writer and (possibly) multiple readers that operate without any coordination. The following sources from the real-time domain address the semantics of state-based communication.

The "standard for space and time partitioned safety-critical avionic real-time operating systems" (ARINC 653) [1] describes two concepts for inter-partition communication; queuing ports and sampling ports. Queuing ports are similar to asynchronous message passing, and sampling ports are what we are concerned with in this paper.

In [10, sect.4.3.4] Kopetz discusses time-triggered messages – the alternative to event-based messages – and he writes: *"The semantics of state messages is similar to the semantics of a program variable that can be read many times without consuming it. Since there are no queues involved in state message transmissions, queue overflow is no issue. [...] State messages support the principle of independence [...] since sender and receiver can operate at different (independent) rates and there is no means for a receiver to influence the sender."*

In this paper we use the following semantics of state-based communication: State-based communication involves a single writer and one or more readers. Writing and reading of a state value must be performed atomically. Each state variable needs its own data structure and is written or read independently of other state variables. A read should always return the newest completely written state value, and at the global level, multiple concurrent readers should observe the same version of a state value at any given point in time. We refer to this as temporal consistency between readers.

State-based communication should in principle not cause interference between communicating tasks. However, some interference in the form of jitter will be observed due to the critical sections enforcing atomicity or the number of re-reads on a non-blocking algorithm [11]. All mechanisms that establish atomicity inherently cause latency and timing jitter, and minimizing these is a main focus of all work in this area. In practice, this can be tolerated as long as we can find a tight upper bound of the run-time.

State values are typically time-stamped in order to allow a receiving process to check whether the state data has become stale. In this work, we assume that the timestamps are part of the data and that the user writes and checks these timestamps.

The challenge of implementing state-based communication on a truly concurrent multicore platform with distributed memory is to ensure atomicity of read and write transactions. We also note that temporal consistency is an ideal that may not be 100 % feasible in distributed algorithms due to different communication latencies towards different readers. In practice, some jitter can be tolerated.

6.3.2 Platform Model and Evaluation Platform

This paper considers a time-predictable multicore platform, where all processing cores are connected to a globally shared off-chip memory, a NoC, and a local scratchpad memory (SPM). Furthermore, we consider that the network-on-chip provides data transfer capabilities, such that each core can push data from its local SPM to a remote SPM.

To generalize our work and to avoid benchmarking features specific to the evaluation platform, we implement synchronization in software where needed. Implementing the communication primitives on a platform with hardware support for synchronization will allow optimization of the synchronization.

We evaluate our work on the open-source T-CREST platform [22], which is a multicore platform developed specifically to be time-predictable. However, the presented algorithms and analysis can easily be adapted to other multicore platforms that include a message passing NoC that can guarantee latency and bandwidth and an SPM in each node. Examples of such platforms include the Kalray MPPA processor series [5], which uses network calculus [16] to find guarantees, or the IDAMC [18] NoC, which uses virtual channel buffers and Back Suction [4] to find guarantees.

In the T-CREST platform, each node contains a Patmos [20] core, an instruction cache, a data cache, and a local SPM under software control. The platform has two NoCs, one that provides access to a shared memory via a memory arbiter [21], and Argo [9] that supports inter-core communication. Both NoCs use TDM to guarantee latency and bandwidth.

The Argo packet-switched NoC implements end-to-end virtual circuits driven by direct memory access (DMA) controllers in the sending platform nodes. A core can set up a DMA controller to push a block of data from the local SPM into the SPM of a remote core. The Argo NoC uses a static TDM schedule for routing communication flows through routers and links. The static schedule is generated by the off-line scheduler [23] for the specified application.

6.3.3 Application Model

Our system contains a set $\mathcal{T} = \{\tau_1, \tau_2, ..., \tau_n\}$ of n periodic tasks that communicate via a set $\mathcal{F} = \{f_1, f_2, ..., f_l\}$ of l state-based communication flows.

Each communication flow $f_j \in \mathcal{F}$ is characterized by a tuple (w_j, r_j, m_j, b_j), where w_j is the task that writes to f_j, r_j is the task that reads from f_j, m_j is the size of a state message, and b_j is the guaranteed bandwidth through f_j. It holds that $w_j \in \mathcal{T}$, $r_j \in \mathcal{T}$, and $w_j \neq r_j$.

Each task $\tau_i \in \mathcal{T}$ is characterized by a tuple (T_i, C_i, p_i), where T_i is the period of the task, C_i is the WCET of the task, and p_i is the statically assigned processor core on which the task is executed on using partitioned scheduling. The k^{th} release of a task τ_i is called a job τ_i^k and may experience some release jitter J_i, introduced by the scheduling, e.g., for a static priority preemptive scheduling policy jitter is upper bounded by $T_i - C_i$.

For simplicity, we only consider tasks that execute on different cores to communicate. Communicating tasks that execute on the same core can easily be added to any of the algorithms we propose, because all our algorithms work for true concurrent systems. To reduce the inter-core interference and simplify the analysis of end-to-end properties, we assume that the critical sections of tasks are non-preemptible.

In our system, a job of a task only performs a single write or read to one communication flow, but a job is allowed to communicate through multiple communication flows.

We define the WCCD D_j of state-based communication flow f_j as the worst-case separation time between the start of the write primitive and the end of the first instance of the read primitive that reads the new value. This definition makes D_j independent of the algorithm and ensures that the tasks of each core can be schedules independently of the tasks executing on other cores.

We assume that the reads and writes of communication flows within a task τ_i are executed unconditionally, such that the reads and writes are executed in the same sequence for every job. Therefore, each job is split into a sequence of phases. These phases can have one of three types: a reading phase, a computation phase, or a writing phase. Thus, the total WCET C_i of a job can be decomposed into a sequence of phases, where each element of the sequence is the WCET of that execution phase.

6.4 Communication Algorithms

We present five state-based communication algorithms and their implementations in on-chip distributed memory. In the following, we use the terms writer managed memory and reader managed memory to denote the memory allocated in the local SPM of the writer and the local SPM of the reader, respectively.

The following subsection describe common considerations of the algorithms, the five algorithms one by one, and how the algorithms can be extended to multiple readers also referred to as multi-casting.

6.4.1 Common Considerations

To ensure atomicity of the state-based communication, it is necessary that reads and writes are performed mutually exclusively, while concurrent reads are allowed. To guarantee mutual exclusion between the writer and the readers, the first four algorithms use a lock and the fifth algorithm uses a queue.

For the four algorithms that use a lock, the length of the critical section of the read or write primitives directly impact the synchronization delay experienced by the other end of the communication. If the read or write functions move a new state value while they are in their critical sections, then the WCET of critical sections are comparable to the WCET of whole function.

Communication between the processing cores of the platform can be executed in two ways: the writer writes/pushes data to a reader or a reader reads/pulls data from the writer. These communication paradigms are called push and pull communication, respectively. Push communication is a single unidirectional transfer of data, whereas pull communication involves a request followed by a transfer of data in the opposite direction, causing higher latency. If we implement state-based communication using push communication, all messages are pushed through the network even though a message may not be read before it is overwritten by the next message. If we implement state-based communication using pull communication, the slave pulls a message through the network even though is was not updated since the last time it was written.

We use push communication to implement the five algorithms, because of the lower latency of push communication. With push communication the writer transfers the state value from its local SPM across the NoC to the local SPM of the reader and the reader copies out the state value from its local SPM.

The writer and reader can transfer or copy the state value, respectively, inside or outside their critical sections. The four algorithms that use a lock represent the four combinations of inside or outside the critical section of the reader and writer.

6.4.2 Algorithm 1: Single Shared Buffer and a Lock

A common practice of implementing an atomically updated shared variable uses a single buffer that is protected by a lock. We implement the shared buffer by allocating it in the processor-local SPM of the reader. The write operation acquires the lock and transfers the new state value to the allocated buffer and then it releases the lock. The read operation acquires the lock and reads the newest state value from the allocated buffer, before it releases the lock.

This is probably the simplest way of implementing state-based communication. However, the implementation has a long critical section because the state value is copied inside the critical section.

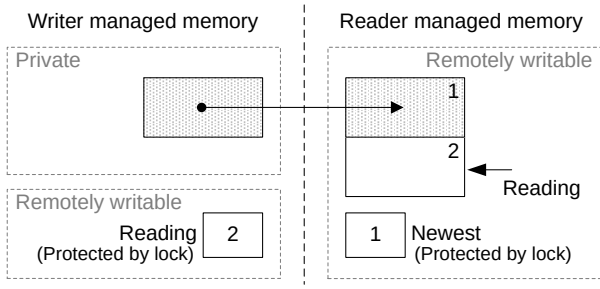


Figure 6.1: The memory layout of the double buffering schemes with an example scenario.

6.4.3 Algorithm 2 & 3: Double Buffering and a Lock

To reduce the interference between reading and writing, we reduce the length of the critical sections of the read and write primitives. By using two shared buffers, we can move the copying or the transferring of the state value out of the critical section of either the writer or the reader. Figure 6.1 shows the memory layout of the double buffering schemes with two shared buffers and two pointers.

An algorithm that copies the state value outside the critical section of the writer needs a `Newest` pointer that points to the newest completely written value. The writer alternates between writing a state value to one or the other buffers in the reader SPM. When the writer has completed writing a state value to a buffer it acquires the lock, moves the newest pointer to the newly written buffer, and then releases the lock. In this algorithm, the reader acquires the lock, reads the buffer that the newest pointer points to, and releases the lock. We refer to this algorithm as algorithm 2 – *reader blocking*.

An algorithm that copies the state value outside the critical section of the reader needs a second `Reading` pointer that points to the state value that the reader is reading. The writer writes a new state value to the buffer that the reader is not reading. The writer acquires the lock, transfers the state value to the buffer that the reader is not reading, updates the newest pointer to point to the newly written value, and releases the lock. The reader acquires the lock, updates the reading pointer to the buffer that the newest pointer is pointing at, releases the lock, and copies the new state value out of that buffer. We refer to this algorithm as algorithm 3 – *writer blocking*.

6.4.4 Algorithm 4: Triple Buffering and a Lock

By using three buffers, we can remove the copying of the state value from the critical section of the reader and the writer. Figure 6.2 shows a snapshot of the algorithm considering three shared buffers. The writer managed memory

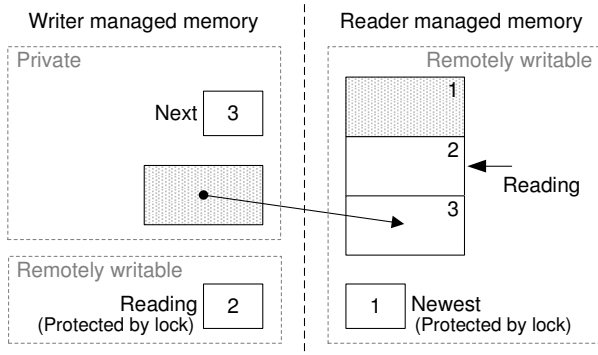


Figure 6.2: An example of a buffering scheme with three buffers for one writer and one reader that only acquire the lock once per operation. The gray buffer in the writer managed memory is transfered to buffer 3 in the reader managed memory. The gray buffer in the reader managed memory is the newest completely written value.

contains the private `Next` variable, the writ buffer, and the remotely writable `Reading` variable. The reader managed memory contains the remotely writable `Newest` variable and the three state value buffers.

In Figure 6.2, the reader is reading from buffer two while the writer has completed writing to buffer one and started to write to buffer three. The access to the three buffers in the reader managed memory is controlled by the shared variables `Newest` and `Reading` that are protected by a lock. The `Next` variable is incremented in a circular manor to point to the next buffer that the writer should transfer a state value to. If the `Next` value is incremented to point to the same buffer as the `Reading` variable, it is incremented again.

The following two enumerated lists describe the steps involved for a write and a read operation. A write operation:

1. Transfer the state message to the buffer pointed to by the variable `Next`.
2. Acquire the lock.
3. Update the variable `Newest` to point to the newly written state buffer.
4. Read the variable `Reading`
5. Update the variable `Next` to point to the buffer that is free. The `Next` buffer is the one that does not contain the newest state and that is not being read.
6. Release lock

A read operation:

1. Acquire the lock
2. Read the variable `Newest`.
3. Update the variable `Reading` to the value of `Newest`.
4. Release lock
5. Read the state message from the buffer pointed to by the variable `Newest`.

A drawback of the presented buffering scheme is that the memory footprint increases. For large message sizes, this buffering scheme might not be feasible in practice.

6.4.5 Algorithm 5: Message Passing Queue

A message passing queue can be used as a solution to the readers/writers problem, if we can find the upper bound on the number of elements in the queue that are needed to avoid overflow. With a queue, the writer writes to the next free buffer and the reader can read the newest value by dequeuing all available elements, only keeping the most recent one.

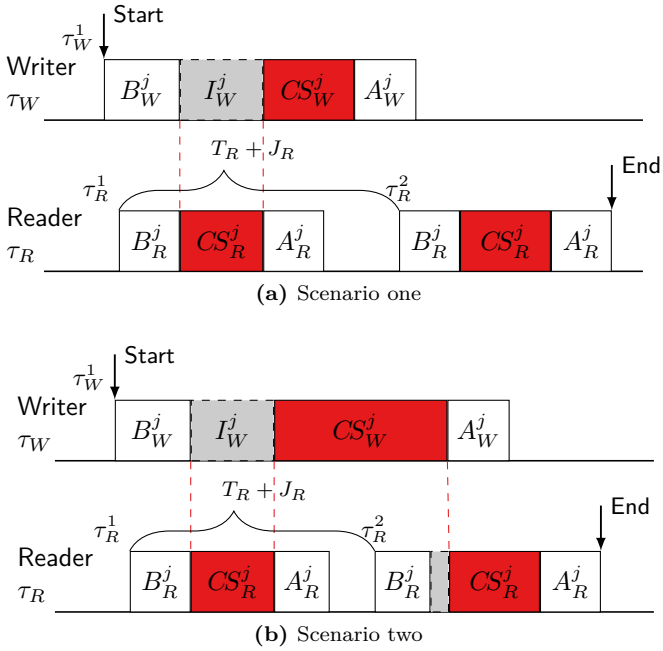
To find the upper bound on the number of elements that are needed to avoid overflow, we need to know the maximum write rate and the minimum read rate.

For periodic tasks, the rate is the number of writes or reads of the state value during the task period over the period. The number of elements needed in the queue is the ratio of the production rate over the consumption rate plus one extra buffer to account for a possible offset in the release time of the writer.

A drawback of the message passing queue is that the memory footprint increases with the period ratio of the writer and reader. When the reader has a shorter period than the writer, we only need two buffers at the reader side. Otherwise, more buffers are needed at the reader side. For large message sizes and a reader with a long period, this buffering scheme might not be feasible in practice.

6.4.6 Extending to Multi-casting

We can extend the five algorithms to multi-casting by allocating a copy of the reader managed memory in the local SPM of each reader and by issuing write transfers through communication flows for each reader, or by performing a multi-cast operation, if the hardware supports this. In the T-CREST platform the individual write transfers can be setup to transfer the new state value in parallel. In a multi-casting implementation, we can use a task-fair read-write lock [2] to



(a) Scenario one

(b) Scenario two

Figure 6.3: The two scenarios that lead to the WCCD. In both cases the condition that leads to the worst-case is when reads and writes are aligned exactly so that the critical section of a read blocks the critical section of a write.

reduce the worst-case synchronization delay, by allowing multiple readers to enter their critical sections at the same time.

To ensure a starvation free solution to the readers/writers problem, the lock needs to grant access to the lock requesters in the same order as the requests are made. The temporal consistency property is fully satisfied for the four algorithms using a lock, because the lock enforces the strict ordering of accesses. Thus, all reads will return the new state value.

The temporal consistency property of the algorithm that uses a queue is not completely satisfied. There can be jitter on the completion of a write, observed locally at the readers. This jitter is caused by variable network latencies and variable start times of the individual DMA transfers. This jitter can be reduced if the hardware platform supports multi-casting data transfers.

6.5 Worst-case Communication Delay

This section presents an analysis of the WCCD of the five algorithms that is independent of how tasks are schedules. The temporal alignment of the writer and reader jobs that cause the WCCD D_j, occurs when a job of the reader causes the maximum interference on a job of the writer.

The alignment of the WCCD is illustrated in Figure 6.3a. If τ_R^1 would be released later in time, then τ_W^1 would instead block τ_R^1 and τ_R^1 would read the new value. If τ_R^1 would be released earlier in time, τ_R^1 would block τ_W^1 for a shorter time and τ_R^2 would be released earlier w.r.t. the beginning of τ_W^1. In both cases, the communication delay is shorter than the WCCD.

If the second job of the reader task τ_R^2 is interfered by a job of the writer task, then τ_R^2 will read the new value of the writer task. This scenario is illustrated in Figure 6.3b. The following two subsections present the WCCD formulas for the presented algorithms.

6.5.1 Algorithm 1 – 4: Shared Buffers and a Lock

For the four presented algorithms that use a lock to protect one, two or three buffers, we model the read and the write functions that communicate through flow f_j as five variables, where the subscript S denotes the index R of the reader task τ_R or the index W of the writer task τ_W.: (1) B_S^j is the WCET of the preamble *Before* the critical section, (2) I_S^j is the worst-case synchronization interference, (3) CS_S^j is the WCET of the critical section, (4) A_S^j is the WCET of the postamble *After* the critical section, and (5) m_j is the message size of the state-based value. The superscript j of the variables denotes the flow index. The read and write primitives inherit their period T_i from the calling task.

The worst-case traversal time (WCTT) of transmitting the state value through the NoC is included in one of the B_S^j, CS_S^j, or A_S^j, depending on the algorithm. The WCTT of transmitting the state value through the NoC is proportional to the bandwidth b_j and is in the case of our NoC, calculated from the TDM schedule.

Figure 6.3a shows the most common scenario that leads to the WCCD. In case T_R is short enough that the critical section of τ_W^1 can interfere with the critical section of τ_R^2, then it is the scenario in Figure 6.3b that causes the WCCD. To find the WCCD for each scenario in Figure 6.3, we sum up a sequence of the known variables that connect the start and end arrows. The sequence in scenario one is $\{B_W^j, -B_R^j, T_R, J_R, B_R^j, CS_R^j, A_R^j\}$. We subtract the second variable in the sequence $-B_R^j$, because the period $T_R + J_R$ starts B_R^j before B_W^j ends. Observe that if the length of the B_R^j phase of τ_R^1 is decreased while CS_R^j still enforces the maximum interference on τ_W^1, the starting time of $T_R + J_R$ and thus the release time of τ_R^2 is delayed. Therefore, the WCCD

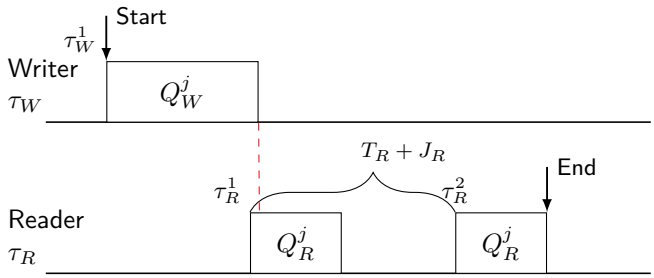


Figure 6.4: The scenario that leads to the WCCD for read and write functions that do not use a lock. The first job of the reader τ_R^1 misses the newly written value.

occurs when the execution time of B_R^j of τ_R^1 is the best-case execution time $\hat{B}_R^j$. We show the WCCD for scenario one:

$$D_j = B_W^j - \hat{B}_R^j + T_R + J_R + B_R^j + CS_R^j + A_R^j \tag{6.1}$$

We assume that $\hat{B}_R^j \ll T_R$, therefore we set $\hat{B}_R^j = 0$. This is a safe underestimation of the best-case execution time that leads to a safe overestimation WCCD. We also find the sequence of known variables in scenario two and by reordering to resemble (6.1), we get:

$$D_j = B_W^j + I_W^j + CS_W^j + CS_R^j + A_R^j \tag{6.2}$$

To unify (6.1) and (6.2) we take the maximum of the two formulas. We show the formula for the WCCD:

$$D_j = B_W^j + \max(T_R + J_R + B_R^j, I_W^j + CS_W^j) + CS_R^j + A_R^j \tag{6.3}$$

The write functions do not return until after the complete transfer of the state value where the new value is available in the SPM of the reader task. Therefore, the message size m_j changes the WCET of the primitives, depending on which algorithm is used.

6.5.2 Algorithms 5: Message Passing Queue

The presented algorithm that implements state-based communication with a message passing queue does not use a lock and therefore there is no synchronization interference. We model the read and write primitives for flow f_j as the WCETs Q_R^j and Q_W^j, respectively. When Q_W^j has completed, the new state value is completely written into the reader SPM. For this algorithm, the temporal alignment that causes the WCCD is when τ_R^1 just misses the new state value before it starts copying out the previously newest value, such that it is τ_R^2

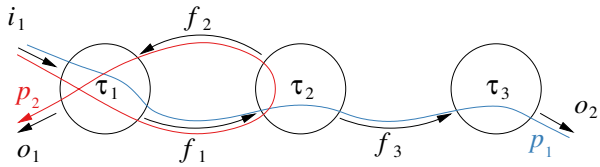


Figure 6.5: An example application with three tasks, three communication flows, and two paths for end-to-end latency analysis.

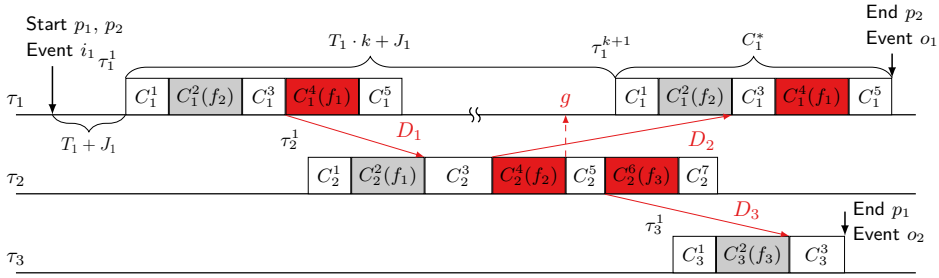


Figure 6.6: The three tasks of our application with execution phases and WCCD. Red execution phases are writes and grey execution phases are reads.

that reads the newest value. This is illustrated in Figure 6.4. The WCCD for this scenario is:

$$D_j = Q_W^j + T_R + J_R + Q_R^j \tag{6.4}$$

The value of Q_R^j depends on the number of elements that are needed in the queue to avoid overflow.

6.6 Maximum End-to-end Latency

End-to-end latency is the time from when an input event occurs until the resulting output event occurs. This section uses the WCCDs derived in the previous section, to analyze the end-to-end or input-to-output latency of a small example with three tasks that communicate via state-based communication. The example contains a straight path and a path that loops back to a previous task.

Figure 6.5 shows the example that comprises three periodic tasks τ_1, τ_2, and τ_3, one input event i_1, two output events o_1 and o_2, three communication flows f_1, f_2, and f_3, and two paths p_1 and p_2 that indicate the propagation of the input value to the outputs. A path is defined by the triggering input event, the involved tasks and communication flows, plus the resulting output event. A detailed timing diagram of the three tasks, including their execution

phases and their mutual communication, is illustrated in Figure 6.6 using the notation introduced in Section 6.3.3. The tasks are periodic and in the example, task τ_1 (identified by the subscripts) has five execution phases (identified by the superscripts). The read primitive and the write primitive are connected by an arrow representing the WCCD D_j found in Section 6.5, where the red execution phases are write primitives and the gray execution phases are read primitives. The f_j in parenthesis of the colored execution phases is the communication flow that the state values are sent across.

Below we analyze the end-to-end latency of the two paths p_1 and p_2 from the example in Figure 6.5. We represent the paths as event chains of tasks and flows, such that $p_1 = \{i_1, \tau_1, f_1, \tau_2, f_3, \tau_3, o_2\}$ and $p_2 = \{i_1, \tau_1, f_1, \tau_2, f_2, \tau_1, o_1\}$. To calculate the end-to-end latency of either path, we add the WCETs of each phase from the start of the path to the end of the path. For path p_1 we get the end-to-end latency D_{p_1}:

$$D_{p_1} = T_1 + J_1 + C_1^1 + C_1^2 + C_1^3 + D_1 + C_2^3 + C_2^4 + C_2^5 + D_3 + C_3^3$$

For path p_2, we use the knowledge that τ_1 is periodic and calculate the maximum number of periods, as Lauer et al. [15] shows on a simpler task model, before τ_1 gets the updated state value from τ_2.

$$\begin{aligned} D_{p_2} &= T_1 + J_1 + T_1 \cdot k + J_1 + C_1^* \\ &= T_1 \cdot (k+1) + 2J_1 + C_1^* \end{aligned}$$

C_1^* is the total WCET of all the phases of τ_1 and k is the number of periods from τ_1^1 to τ_1^{k+1} that observes a change as a result of the input event i_1 through the flow f_2. We calculate k as the ceiling of the path from the release of τ_1^1 to g divided by the period T_1 of τ_1, where g is the time when the new value of f_2 can be read by τ_1. The formula for k is:

$$k = \left\lceil \frac{C_1^1 + C_1^2 + C_1^3 + D_1 + C_2^3 + C_2^4}{T_1} \right\rceil \tag{6.5}$$

Depending on which algorithm of state-based communication the developer chooses, either the formulas in (6.3) or (6.4) should be inserted in place of the WCCD of the flows.

These formulas are valid for any task schedule where all tasks meet their deadline. If we include a concrete schedule, it may be possible to reduce the maximum end-to-end latency of the paths with the holistic schedulability analysis presented by Tindell and Clark [25].

6.7 Worst-case Evaluation

This section describes the evaluation setup and the evaluation of the five algorithms. As we consider real-time systems, we use static WCET analysis for the performance comparison instead of average-case measurements.

6.7.1 Evaluation setup

For this evaluation, we assume a 9 core platform, where the code for the primitives is stored in the instruction SPM. In the 9 core platform, the bandwidth towards main memory is divided equally between the 9 cores and the guaranteed-service of the NoC is setup such that all cores can send to all other cores with equal bandwidth. We refer to this NoC schedule as an all-to-all schedule. The guaranteed service to each core corresponds to a guaranteed minimum bandwidth of one 8-byte packet every 36 clock cycles (cc) and a guaranteed maximum latency of 45 cc for 8 bytes and 297 cc for 64 bytes.

We find the WCET of the communication primitives with the aiT tool from AbsInt [6], which supports the Patmos processor. In the source code of the communication primitives, there are a number of busy-wait `while` loops that continuously loop until certain events that are time-bounded happen, such as the completion of a DMA transfer. The worst-case waiting time of a DMA transfer can be calculated based on the size of the transfer and the bandwidth of the communication flow towards the receiver. The worst-case wait time divided by the WCET of one iteration of the `while` loop is equal to the maximum loop bound of that `while` loop. For each data point shown in the following plots, we found the WCET of one iteration of all the `while` loops. Based on the maximum waiting time and the WCET of each loop iteration, we can calculate the loop bounds of each `while` loop and pass them to the tool.

The WCET numbers that we show in the following subsections include the code for the locking functions `acquire_lock()` (240 cc) and `release_lock()` (82 cc). These numbers do not account for the interference from other cores that try to take the lock. The application designer needs to add the interference of the other threads holding the lock to the length of the critical sections during the schedulability analysis.

The lock that we use for the results is Lamport's Bakery [13] algorithm using the on-chip distributed memory. The Bakery algorithm is well-suited for implementation in distributed memory, because the variables can be laid out such that it uses local-only spinning and remote writes.

6.7.2 Algorithm 1 – 4: Shared Buffers and a Lock

Figure 6.7 shows the WCET of the critical sections, read_cs and write_cs, and the whole read and write functions, read and write, for algorithm 1 – 4 as a function of the message size.

Figure 6.7a shows the WCET of algorithm 1. We see that the critical section of the write primitive is longer than the critical section of the read primitive. This is due to the fact that the network bandwidth of the all-to-all schedule is lower than the bandwidth between the local SPM and the processor. Furthermore, the gaps between the WCET of the whole functions and their

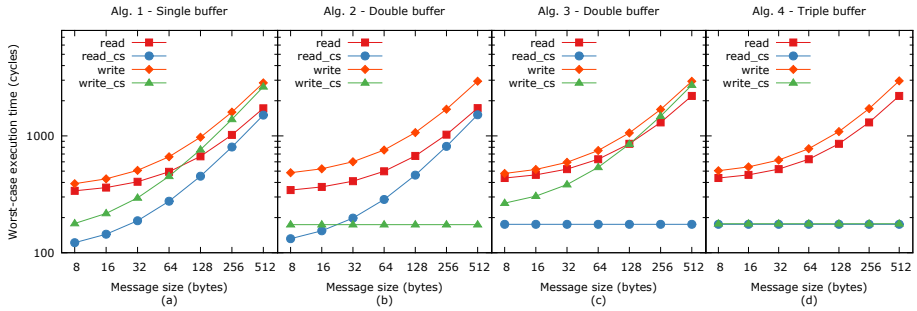


Figure 6.7: The WCET of the read and write functions of algorithm 1 to 4 and their critical sections, as a function of the message size.

critical sections are the 240 cc overhead of the `acquire_lock()` function, which is constant across all message sizes.

Figure 6.7b shows the WCET of algorithm 2. We see that the critical section of the write function is constant across all message size, at the cost of a slightly higher WCET of the whole write primitive. The WCET of the read function is very similar to that of algorithm 1. The read function of algorithm 2 contains an extra load from the local SPM and a comparison, to determine which buffer to read from.

Figure 6.7c shows the WCET of algorithm 3. The critical section of the read function is constant cross all message sizes and the critical section of the write function is slightly increased, compared to the critical section of the write function in algorithm 1. Compared to algorithm 2, the WCET of the whole write function is lower and of the whole read function is higher.

Figure 6.7d shows the WCET of algorithm 4. The WCET of the critical sections of the read and write primitives are constant across the different message sizes. The length of the critical sections of the read and the write primitive are similar, because they both contain one 8-byte write through the network. Furthermore, we see that the difference between the critical section and the overall WCET of the communication primitive scales linearly with the message size.

For the four algorithms, the write functions have higher WCETs than the read functions, because the bandwidth of the NoC transfer is lower than the bandwidth towards the local SPM. The algorithms with shorter critical sections considerably reduce the interference between the read and write functions, at the cost of slightly higher WCETs. Furthermore, a NoC TDM schedule with higher bandwidth from the writer to the reader, will reduce the gap between the WCET of the whole write function and the read function.

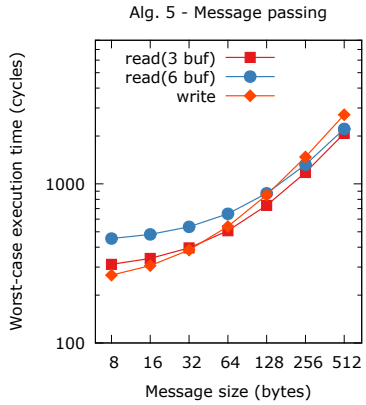


Figure 6.8: The WCET of the read and write functions, with the queuing algorithm, as a function of the size of a sample.

6.7.3 Algorithm 5: Message Passing Queue

The queuing algorithm does not have any locking and therefore no critical sections. Figure 6.8 shows the WCET of the write primitive and the WCET of the read primitive with 3 and 6 elements in the queue. With 3 buffers, the algorithm supports that the writer writes twice as fast as the reader reads. With 6 buffers the ratio is 5-times faster writes.

The number of buffers in the queue changes the WCET of the read primitive. In the worst-case, the reader needs to dequeue all the elements of the buffer and then return the last successfully dequeued message. The WCET of the read primitive increases with the number of buffers, but the reading of the message becomes the dominating factor in the WCET as the message size grows. Therefore, the WCET of the reads are higher than the WCET of the write for small messages sizes, even though the bandwidth for the remote write transaction is lower than of the local read.

6.7.4 Comparison

Figure 6.9a and 6.9b shows a comparison of the WCET for the write functions and for the read functions, respectively. Figure 6.9c shows the WCCDs D_j minus the period T_R and the jitter J_R of the five algorithms as a function of the message size, as shown in (6.3) and (6.4). We show the numbers without T_R and J_R because these variables are the same for the five algorithms and they are determined by an application. For this WCCD comparison of the five algorithms, we assume that the period T_R is much higher than the WCET of write and read functions, which is scenario one from Figure 6.3a.

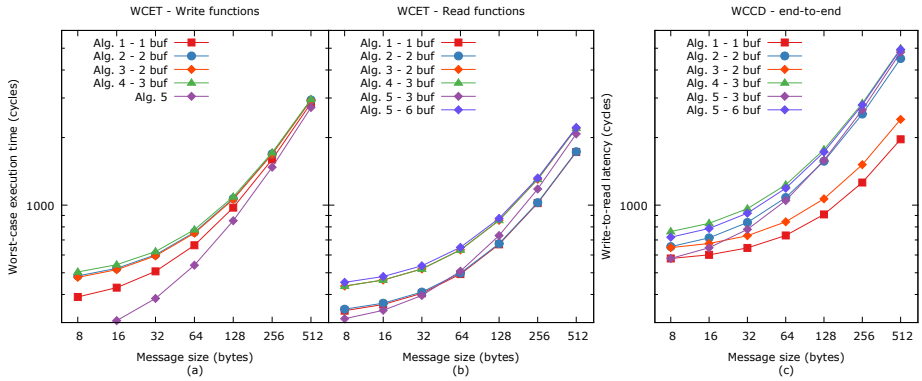


Figure 6.9: The WCET of the read and write functions and the WCCD for all the presented algorithms, as a function of the size of a sample.

There is not one algorithm that performs better than the other algorithms in all scenarios. In the following, we will outline in which scenario each algorithm performs better than the other algorithms.

Algorithm 1, the single buffer algorithm, has the lowest memory footprint, the lowest WCCD, and low WCETs, but it also has the highest synchronization interference due to the long critical sections of the read and write functions.

Algorithm 2, the double buffer algorithm with a short critical section in the write function, also has a low memory footprint and it has low WCET of the read function, but it has a high WCET of the write function.

Algorithm 3, the double buffer algorithm with a short critical section in the read function, has a low WCCD and a low memory footprint, but it has a high WCET and synchronization of the read function due to the long critical section of the write function.

Algorithm 4, the triple buffer algorithm, has a larger memory footprint and high WCETs of the write and read functions, but has lower synchronization interference than the other algorithms that use a lock, due to the short critical sections, but it has the highest WCCD.

Algorithm 5, the message queue with three buffers, is the best solution in terms of WCET and synchronization interference, because it does not suffer any synchronization interference. With six buffers, the WCET of the read function becomes the highest of all the algorithms. The synchronization interference of the write function is low because it does not suffer any synchronization interference. The synchronization interference of the read function suffers mainly from the number of buffers that needs to be dequeued. If the ratio between the writing frequency and the reading frequency is high, then the memory footprint becomes prohibitively high.

Table 6.1: A comparison of the five communication algorithms. For each parameter of comparison the algorithms are ranked from the best (++) to the worst (− −).

	Sync. intf		WCET		WCCD	Memory
Alg.	Write	Read	Write	Read		
1	− −	− −	+	++	++	++
2	− −	+	− −	++	−	+
3	+	− −	−	− −	+	+
4	+	+	− −	− −	− −	−
5	++	++	++	+/− −	−	−(−)

In Figure 6.9c, we see that algorithm 1 and 3 have significantly lower WCCDs than the other algorithms. This is because the preamble of the write functions for algorithm 1 and 3 does not contain the transfer of a state value through the NoC.

In Table 6.1, we summarize the advantages and disadvantages of the five algorithms with the parameters of the WCET and the jitter for the read and write functions, the WCCD, and the memory footprint. For each parameter, we rank each algorithm giving (++) to the lowest and (− −) to the highest of each parameter. Table 6.1 suggests that by increasing the memory footprint and the WCCD, we can lower the synchronization interference. As we see in Table 6.1 the WCETs of the write and read functions are comparable in contrast to the synchronization interference caused by the critical sections as seen in Figure 6.7. In a schedulability analysis, the algorithms that has a high synchronization interference suffer more than the algorithms with a high WCET.

6.8 Conclusion

This paper addressed the implementation of time-predictable state-based communication in multicore platforms for hard real-time systems. The concept of state-based communication is similar to a shared variable that can be written and read atomically.

Aiming for an algorithm that scales better with a growing number of processors, and has low latency and low jitter, this paper proposed and evaluated five algorithms that exploit the scalable message passing NoC and the processor-local memories found in many recent multicore platforms. The evaluation is based on actual hardware and the WCET in clock cycles is obtained using the aiT tool from AbsInt.

We have shown how to calculate the WCCD of the five algorithms and we have outlined in which parameters each algorithms perform better than the others. For the five algorithms, a reduction in the synchronization interference results in an increase of the WCCD.

Acknowledgment

The work presented in this paper was funded by the Danish Council for Independent Research | Technology and Production Sciences under the project RTEMP[19], contract no. 12-127600. The work is open source and the full T-CREST tool chain can be downloaded from GitHub (`https://github.com/t-crest/`) and built under Ubuntu.

Bibliography

[1] ARINC 653. Avionics application software standard snterface – Part 1: Required services, 2010.

[2] B. Brandenburg and J. Anderson. Spin-based reader-writer synchronization for multiprocessor real-time systems. *Real-Time Systems*, 46(1): 25–87, 2010. ISSN 0922-6443. doi: 10.1007/s11241-010-9097-2. URL `http://dx.doi.org/10.1007/s11241-010-9097-2`.

[3] P. J. Courtois, F. Heymans, and D. L. Parnas. Concurrent control with "readers" and "writers". *Commun. ACM*, 14(10):667–668, Oct. 1971. ISSN 0001-0782. doi: 10.1145/362759.362813. URL `http://doi.acm.org/10.1145/362759.362813`.

[4] J. Diemer and R. Ernst. Back suction: Service guarantees for latency-sensitive on-chip networks. In *ACM/IEEE International Symposium on Networks-on-Chip (NOCS)*, pages 155–162, May 2010. doi: 10.1109/NOCS.2010.38.

[5] B. Dupont de Dinechin, D. van Amstel, M. Poulhiès, and G. Lager. Time-critical computing on a single-chip massively parallel processor. In *Proc. Design, Automation and Test in Europe (DATE)*, pages 97:1–97:6, 2014. ISBN 978-3-9815370-2-4. URL `http://dl.acm.org/citation.cfm?id=2616606.2616725`.

[6] C. Ferdinand, R. Heckmann, M. Langenbach, F. Martin, M. Schmidt, H. Theiling, S. Thesing, and R. Wilhelm. Reliable and precise WCET determination for a real-life processor. In T. A. Henzinger and C. M. Kirsch, editors, *EMSOFT*, volume 2211 of *Lecture Notes in Computer Science*,

pages 469–485. Springer, 2001. ISBN 3-540-42673-6. URL http://link.springer.de/link/service/series/0558/bibs/2211/22110469.htm.

[7] O. Gangwal, A. Rădulescu, K. Goossens, S. González Pestana, and E. Rijpkema. Building predictable systems on chip: An analysis of guaranteed communication in the aethereal network on chip. In P. van der Stok, editor, *Dynamic and Robust Streaming in and between Connected Consumer-Electronic Devices*, volume 3 of *Philips Research*, pages 1–36. Springer Netherlands, 2005. ISBN 978-1-4020-3453-4. doi: 10.1007/1-4020-3454-7_1.

[8] L. S. Indrusiak. End-to-end schedulability tests for multiprocessor embedded systems based on networks-on-chip with priority-preemptive arbitration. *Journal of Systems Architecture - Embedded Systems Design*, 60(7): 553–561, 2014. URL http://dx.doi.org/10.1016/j.sysarc.2014.05.002.

[9] E. Kasapaki, M. Schoeberl, R. B. Sørensen, C. T. Müller, K. Goossens, and J. Sparsø. Argo: A real-time network-on-chip architecture with an efficient GALS implementation. *Very Large Scale Integration (VLSI) Systems, IEEE Transactions on*, PP, 2015. doi: 10.1109/TVLSI.2015.2405614. URL http://www.jopdesign.com/doc/argo-jnl.pdf.

[10] H. Kopetz. *Real-Time Systems*. Kluwer Academic, Boston, MA, USA, 1997.

[11] H. Kopetz and J. Reisinger. The non-blocking write protocol nbw: A solution to a real-time synchronization problem. In *Real-Time Systems Symposium, 1993., Proceedings.*, pages 131–137, Dec 1993. doi: 10.1109/REAL.1993.393507.

[12] O. Krieger, M. Stumm, R. Unrau, and J. Hanna. A fair fast scalable reader-writer lock. In *Proc. Int. Conference on Parallel Processing (ICPP)*, volume 2, pages 201–204, 1993. doi: 10.1109/ICPP.1993.21.

[13] L. Lamport. A new solution of dijkstra's concurrent programming problem. *Commun. ACM*, 17(8):453–455, Aug. 1974. ISSN 0001-0782. doi: 10.1145/361082.361093. URL http://doi.acm.org/10.1145/361082.361093.

[14] L. Lamport. Concurrent reading and writing. *Commun. ACM*, 20(11): 806–811, Nov. 1977. ISSN 0001-0782. doi: 10.1145/359863.359878. URL http://doi.acm.org/10.1145/359863.359878.

[15] M. Lauer, J. Ermont, F. Boniol, and C. Pagetti. Latency and freshness analysis on IMA systems. In *Proc. IEEE Conference on Emerging Technologies Factory Automation (ETFA)*, pages 1–8, Sept 2011. doi: 10.1109/ETFA.2011.6059017.

[16] J.-Y. Le Boudec. Application of network calculus to guaranteed service networks. *Information Theory, IEEE Transactions on*, 44(3):1087–1096, May 1998. ISSN 0018-9448. doi: 10.1109/18.669170.

[17] J. M. Mellor-Crummey and M. L. Scott. Scalable reader-writer synchronization for shared-memory multiprocessors. In *Proc. ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPOPP)*, pages 106–113. ACM, 1991. ISBN 0-89791-390-6. doi: 10.1145/109625.109637. URL `http://doi.acm.org.globalproxy.cvt.dk/10.1145/109625.109637`.

[18] B. Motruk, J. Diemer, R. Buchty, R. Ernst, and M. Berekovic. IDAMC: A many-core platform with run-time monitoring for mixed-criticality. In *Proc. IEEE Int. Symposium on High-Assurance Systems Engineering (HASE)*, pages 24–31, 2012. doi: 10.1109/HASE.2012.19.

[19] RTEMP project page, 2013. URL `http://rtemp.compute.dtu.dk/`. Available online at `http://rtemp.compute.dtu.dk/`.

[20] M. Schoeberl, P. Schleuniger, W. Puffitsch, F. Brandner, C. W. Probst, S. Karlsson, and T. Thorn. Towards a time-predictable dual-issue microprocessor: The Patmos approach. In *First Workshop on Bringing Theory to Practice: Predictability and Performance in Embedded Systems (PPES 2011)*, pages 11–20, Grenoble, France, March 2011. URL `http://www.jopdesign.com/doc/patmos_ppes.pdf`.

[21] M. Schoeberl, D. V. Chong, W. Puffitsch, and J. Sparsø. A time-predictable memory network-on-chip. In *Proceedings of the 14th International Workshop on Worst-Case Execution Time Analysis (WCET 2014)*, pages 53–62, Madrid, Spain, July 2014. doi: 10.4230/OASIcs.WCET.2014.53. URL `http://www.jopdesign.com/doc/memnoc.pdf`.

[22] M. Schoeberl, S. Abbaspour, B. Akesson, N. Audsley, R. Capasso, J. Garside, K. Goossens, S. Goossens, S. Hansen, R. Heckmann, S. Hepp, B. Huber, A. Jordan, E. Kasapaki, J. Knoop, Y. Li, D. Prokesch, W. Puffitsch, P. Puschner, A. Rocha, C. Silva, J. Sparsø, and A. Tocchi. T-CREST: Time-predictable multi-core architecture for embedded systems. *Journal of Systems Architecture*, 61(9):449–471, 2015. ISSN 18736165, 13837621. doi: 10.1016/j.sysarc.2015.04.002.

[23] R. B. Sørensen, J. Sparsø, M. R. Pedersen, and J. Højgaard. A metaheuristic scheduler for time division multiplexed network-on-chip. In *Proc. IEEE Workshop on Software Technologies for Future Embedded and Ubiquitous Systems (SEUS), 2014*. IEEE, 2014.

[24] P. G. Sorenson and V. C. Hamacher. A real-time system design methodology. *INFOR. Canadian Journal of Operational Research and Information Processing*, 13(1):1–18, 1975. ISSN 03155986, 19160615.

[25] K. Tindell and J. Clark. Holistic schedulability analysis for distributed hard real-time systems. *Microprocessing and Microprogramming*, 40(2-3):117–134, 1994. ISSN 18787061, 01656074. doi: 10.1016/0165-6074(94)90080-9.

Chapter 7

Conclusions

The previous five chapters investigate time-predictable inter-core communication on a multicore platform with a statically scheduled TDM NoC. This chapter summarizes the findings of these five chapters and identifies areas for future work.

7.1 Summary of Findings

This thesis contributes in the three research areas: reconfigurable time-division multiplexing networks-on-chip, static time-division multiplexing scheduling, and time-predictable inter-core communication. The TDM NoC uses statically generated TDM schedules to provide GS VC that supports the time-predictable inter-core communication. This section summarizes the findings of each of the three research areas and the overall findings in the following four subsections.

7.1.1 Reconfigurable Time-Division Multiplexing Network-on-Chip

We have extended the Argo 1.0 TDM NoC with a new generation of the NI architecture that increases the flexibility of the hardware compared to the previous Argo version. We have increased the flexibility by adding the possibility of remotely writing the configuration tables in the NI. Additionally, we have added support for instantaneous reconfiguration of the VCs. The NI hardware

architecture also supports the incremental approach for reconfiguration that is used by previous state-of-the-art TDM NoCs that offer reconfiguration.

The hardware size of our new generation NI and the original Argo router, is less than half the size of architectures that provide similar functionality. The hardware resources are designed to be more flexible and they are able to implement schedules that are more parameterized than what is supported by previous state-of-the-art NoCs. These more parameterized schedules allows the application developer to make efficient use of resources. When we use the compact schedule representation from our new generation NI, the NI can store several schedules in a small schedule table.

7.1.2 Static Time-Division Multiplexing Scheduling

We have designed and implemented a TDM scheduler that can generate schedules for a TDM NoC from a communication pattern. Our TDM scheduler supports a more parameterized class of NoCs and communication patterns than previously published TDM schedulers. The TDM scheduler can generate schedules for custom NoC topologies and with variable packet lengths on VCs. A schedule that can fit the communication pattern of an application better, results in a more efficient use of the NoC hardware resources.

We have shown a method that can shorten the TDM schedules by allocating more bandwidth to the channels of a communication pattern that has the smallest bandwidth requirements. This method reduces the hardware requirements, by reducing the length of a TDM period, with a negligible reduction in the resulting bandwidth.

Our approach of reconfiguration, which needs to store multiple schedules in the schedule table at the same time, benefits twofold from the short TDM schedules. First, they can store a larger number of schedules in less memory of the schedule table. Second, the time to reconfigure can be reduced because the length of the TDM periods are shorter.

7.1.3 Time-Predictable Inter-Core Communication

We have identified and implemented two types of inter-core communication that are used by real-time applications: message passing and state-based communication. For message passing, the consumer needs to consume all produced messages. For state-based communication, the consumer must always read the newest produced message. For both types of inter-core communication, our implementations avoid unnecessary copying of messages, which ensures a low WCET.

We showed that message passing can be implemented efficiently in terms of WCET and worst-case latency of a message and that it matches well with the push communication that the Argo platform provides. Furthermore, the WCET

of the send and receive functions are not dependent on the message size, which simplifies the WCET analysis of a communicating task.

We have implemented five algorithms that provide state-based communication. The results show that the five proposed algorithms each have their advantages and disadvantages and that no one algorithm is better that the others in all scenarios. State-based communication requires a more complex algorithm than message passing, due to the synchronization of atomically updating a state value. With the developed analysis for our communication primitives, tasks that are executed on one core can be scheduled independently of tasks executing on other cores.

Due to the relatively low software overhead, the latencies of sending and receiving state values or messages are greatly affected by the NoC data transfer timings. The NoC data transfer timings are determined by the TDM schedule that is operating in the NoC. We have used an all-to-all schedule for the worst-case numbers in all the results we have shown. If we use an application specific schedule, the transmission time of the NoC data transfer can be greatly reduced, which reduces the total worst-case times even further.

The WCET of the send and receive functions for message passing is low and does not depend on the message size, as the message transfer is decoupled from the program. Whereas, the message transfer of the state-based communication is not decoupled from the program, due to the needed synchronization.

7.1.4 Overall Findings

In this thesis, we investigated a time-predictable NoC from the network interface layer up to the system layer functions in the API. We developed a new resource-efficient and more flexible generation of the NI for the Argo NoC and we showed how to implement two types of time-predictable inter-core communication on this new hardware architecture.

With the increased flexibility of the Argo 2.0 NI, the TDM scheduler can generate TDM schedules that match the requirements of inter-core communication more precisely and thereby reduce the latency of inter-core communication and/or the operating frequency of the NoC.

Compared to NoCs with similar functionality, we implement the task of allocating memory for communication buffers in the software layer rather than fixed hardware buffers, which increases the flexibility of the system.

One use case of the increased flexibility is that the application can allocate buffering space for the inter-core communication algorithms on a per connection basis. Allowing the application to decide which communication algorithm to use, on a per connection basis, depending on which timing properties are desirable.

Furthermore, we added support to the Argo 2.0 for inter-core interrupts and for efficient reconfiguration of the VCs that provide the GS that enable the

time-predictable inter-core communication. These two features are important for supporting a symmetric multiprocessing operating system.

7.2 Future Work

Based on the work carried out in this thesis, we expect the following research areas to bring new important insight into how time-predictable multicore platforms should be build efficiently:

- The added interrupt capabilities of the NI opens up the possibility of adding power management to the processor, such that if the local core is waiting for a remote core, the clock of the processor is gated until an incoming packet causes an interrupt. This clock gating scheme can also be used for hardware accelerators waiting for a request from a remote core. In case the processor clock is gated and the processor is sleeping, the NI has to continue operating to keep track of the TDM schedule.

- Many of the benchmark communication patterns we have used throughout the thesis have a few cores with a large number of outgoing VCs and most cores with a few outgoing VCs. The length of the schedules for these communication patterns are lower bound by the number of outgoing VC for the few cores, we refer to this kind of schedule as being I/O bound. Communication patterns that are I/O bound inherently has a low overall link utilization. We can address this issue by decreasing the number of links. If we can decrease the total number of links without increasing the TDM period of the I/O bound communication patterns, then the link utilization will increase.

  The total number of links can be decreased by reducing the number of links per NI in the network. We propose to investigate how the link utilization and the TDM period changes when four five-ported routers are collapsed into one eight-ported router, as shown in Figure 7.1. This change in topology reduces the number of links by a factor of four, but it only reduces the bisection bandwidth by two. The overall bisection bandwidth could be the same by doubling the link width and reducing the packet lengths. We expect that these changes will reduce the hardware size and the worst-case latency in the network, especially for neighboring cores.

  To investigate this proposal, we need to change the input format that specifies the platform topology and the internal data structure of the scheduler that represents the platform.

- In our new generation of the Argo NI, we introduced the instantaneous reconfiguration capabilities and evaluated how the addition of dedicated configuration VCs extended the TDM period of the benchmarks. When

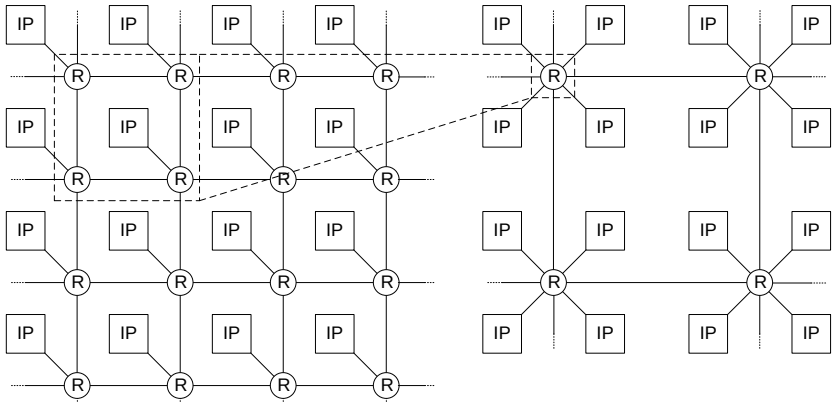


Figure 7.1: Sections of a platform with 16 cores with five-ported routers and with eight-ported routers.

the number of cores in the network increases, the lower bound on the TDM period also increases. If we scale the network up to a size where this lower bound becomes infeasible, we can move the configuration VCs into a dedicated configuration tree network. The hardware size of such a tree network scales linearly in the number of cores, but it restricts that only one core, the master of the network, can trigger a reconfiguration of the network.

- The NoC reconfiguration feature that we introduced enables the NoC to instantaneously switch from one configuration to another in the schedule table. Any VC that exists in the active schedule before and after the reconfiguration must keep its entry in the DMA table to experience uninterrupted GS across the reconfiguration. Generating a static mapping for all VCs, included in the set of configurations that belong to an application, can be modeled as a global register allocation problem. In this model the VCs that persist across one or more reconfigurations are modeled as values that live across a basic block. This is an interesting problem and it should be investigated if a dynamic approach of solving this problem can be made analyzable.